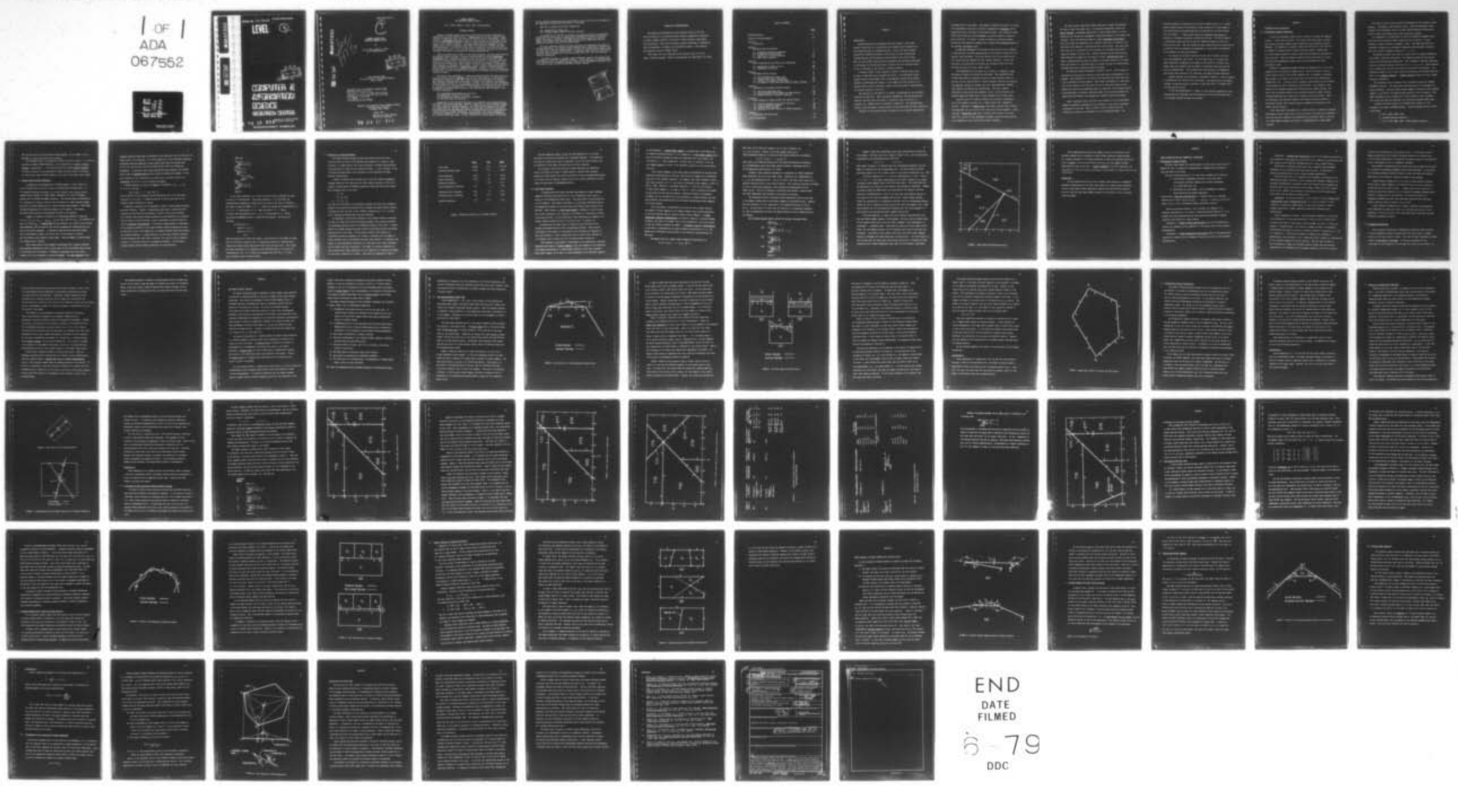
AD-A067 552 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
A DOMAIN STRATEGY FOR COMPUTER PROGRAM TESTING.(U)
AUG 78 L J WHITE, E I COHEN, B CHANDRASEKARAN AFOSR-77-3416
UNCLASSIFIED OSU-CISRC-TR-78-4 AFOSR-TR-79-0444 NL

1 OF 1
ADA
067552



END
DATE
FILMED
6-79
DDC

AFOSR-TR- 79-0444

TECHNICAL REPORT SERIES

LEVEL III

3

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

Approved for public release;
distribution unlimited.

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(OSU-CISRC-TR-78-4)

(3)

A DOMAIN STRATEGY FOR COMPUTER PROGRAM TESTING

by

Lee J. White, Edward I. Cohen
and B. Chandrasekaran

Work performed under
Air Force Office of Scientific Research
Grant 77-3416

Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210

August 1978

A DOMAIN STRATEGY
FOR COMPUTER PROGRAM TESTING

Lee J. White, Edward I. Cohen, and B. Chandrasekaran

EXTENDED ABSTRACT

Computer programs contain two types of errors which have been identified as computation errors and domain errors. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program. A path contains a computation error when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables. A testing strategy has been designed to detect domain errors, and the conditions under which this strategy is reliable are given and characterized. A by-product of this domain strategy is a partial ability to detect computation errors. It is the objective of this study to provide an analytical foundation upon which to base practical testing implementations.

There are limitations inherent to any testing strategy, and these also constrain the proposed domain strategy. One such limitation might be termed coincidental correctness, which occurs when a specific test point follows an incorrect path, and yet the output variables coincidentally are the same as if that test point were to follow the correct path. This test point would then be of no assistance in the detection of the domain error which caused the control flow change. No test generation strategy can circumvent this problem. Another inherent testing limitation has been previously identified as a missing path error, in which a required predicate does not appear in the given program to be tested. Especially if this predicate were an equality, no testing strategy could systematically determine that such a predicate should be present.

The control flow statements in a computer program partition the input space into a set of mutually exclusive domains, each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. The testing strategy generates test points to examine the boundaries of a domain to detect whether a domain error has occurred, as either one or more of these boundaries will have shifted or else the corresponding predicate relational operator has changed. If test points can be chosen within ε of each boundary, the strategy is shown to be reliable in detecting domain errors of magnitude greater than ε, subject to the following assumptions:

(1) coincidental correctness does not occur;
(2) missing path errors do not occur;
(3) predicates are linear in the input variables;
(4) the input space is continuous.

Assumptions (1) and (2) have been shown to be inherent to the testing process, and cannot be entirely eliminated. However, recognition of these potential problems can lead to improved testing techniques. The domain testing method has been shown to be applicable for nonlinear boundaries, but the number of required test points may become inordinate and there are complex problems associated with processing nonlinear boundaries in higher dimensions. The continuous input space assumption is not really a limitation of the proposed testing method, but allows the parameter ε to be chosen arbitrarily small. An error analysis for discrete spaces is available

and the testing strategy has been proved viable as long as the size of the domain is not comparable to the discrete resolution of the space.

Next let us consider two further assumptions:

(5) predicates are simple; and
(6) adjacent domains compute different functions.

If assumptions (5) and (6) are imposed, the testing strategy is considerably simplified, as no more than one domain need be examined at one time in order to select test points. Moreover, the number of test points required to test each domain grows linearly with both the dimensionality of the input space and the number of predicates along the path being tested.

The only completely effective testing strategy is an exhaustive test which is totally impractical. The domain testing strategy offers a substantial reduction in the high cost of computer program testing, and yet can reliably detect a major class of errors which have been characterized. In addition, other types of errors can be detected, such as computation errors and missing path errors, but this detection cannot be guaranteed.

The domain strategy is currently being implemented, and will be utilized as an *experimental facility* for subsequent research. A most important contribution would be to indicate both programming language constructs and programming techniques which are easier to test, and thus produce more reliable software.

PREFACE AND ACKNOWLEDGMENTS

The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization consisting of staff, graduate students, and faculty of many University departments and laboratories. This report describes research undertaken in cooperation with the Department of Computer and Information Science. This research was supported in part by AFOSR 77-3416.

This report was first published in the Infotech State of the Art Report "Software Testing," Infotech International Ltd, Maidenhead, UK (1978).

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Program testing is an inherently practical activity, since every computer program must be tested before any confidence can be gained that the program performs its intended function. Some of the best designed software has required that nearly as much effort be spent planning and implementing the testing process as was invested in the actual coding. What the practitioner needs are better guidelines and systematic approaches in the design of the testing process to replace the ad hoc approach which is now so prevalent in the testing of computer software.

It would be ideal if there existed a "theory of testing" which could be used to rigorously select program test points. The problem has unfortunately proven so intractable that no comprehensive testing theory exists. Research by Goodenough and Gerhart [7] and Howden [8,9] has resulted in an accepted body of theory concerning testing, and has provided a rigorous basis for further research in this area.

The objective of this paper is to present a methodology for the automatic selection of test data. Under appropriate assumptions, this methodology will generate test data which will detect a particular class of errors in a program, viz., "domain errors" as defined by Howden [9]. The proposed methodology is also described in greater detail in Cohen and White [3] and in Cohen [4].

The goal of the testing process is limited to the successful detection of a program error if any exists. Any attempt to identify the error, its cause, or an appropriate correction is properly categorized as debugging, and is beyond the scope of our goal in the testing process. Thus testing is essentially error detection, while debugging is the more difficult process of error correction. Of course, in practice these two activities usually overlap and are frequently combined into a single testing/debugging phase in the software development cycle.

An important assumption in our work is that the user (or an "oracle") is available who can decide unequivocally if the output is correct for the specific input processed. The oracle decides only if the output values are correct, and not whether they are computed correctly. If they are incorrect, the oracle does not provide any information about the error and does not give the correct output values.

The organization of the report is as follows. In Chapter 2, some preliminary concepts are defined and discussed. Some assumptions must be made concerning the language in which the given computer program is written, and the ramifications of certain language constructs are explored. The important concepts of program path and path predicates, together with domains, are defined and characterized. The case of linear predicates is given particular emphasis, since, in that situation, the domains assume the simple form of convex polyhedra in the input space.

Logical errors in a computer program can be viewed as belonging to one of two classes of errors, viz., "domain errors" and "computation errors". Informally, a domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program. A path contains a computation error when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables.

The third chapter rigorously defines these error classes, and explores the ways in which they might arise. The proposed methodology, called the domain strategy, is designed specifically to detect domain errors. In this chapter, we will discuss two fundamental limitations inherent to any finite test strategy. Once such limitation might be termed coincidental correctness. This occurs when the computation for a specific test point is incorrect, but the output value happens to coincide with the correct value. This test point would then be of no assistance in the detection of the domain error which caused the change in control flow. Another inherent testing limitation has been identified by Howden [9], and might be called a missing path error, in which a required predicate does not appear in the given program to be tested. This could result in a situation where no testing strategy can systematically determine that such a predicate should be present.

The domain strategy is examined in Chapters 4 and 5. This strategy is developed by utilizing the structure of the input space corresponding to the program. More specifically, the control flow partitions the input space into a set of mutually exclusive domains. Each domain corresponds to a particular path in the program in the sense that the set of input data points in that domain will cause the corresponding path to be executed. The strategy proposed is path-oriented; in testing a particular path, we are acutally testing the computations performed by the program over a specific input space domain.

Given a particular path, the form of the boundary of the corresponding domain is completely determined by the predicates in the control statements encountered in the path. Thus, an error in such a predicate will be reflected as a shift in the boundary of the corresponding domain. The testing strategy to be described tests a path for domain errors, i.e., detects domain boundary shifts by observing the output values for a finite number of test data having a prescribed geometrical relationship to the entire domain and its boundary. These output values are computed by executing the sequence of assignment statements constituting the path. The method requires no information other than the successfully compiled program for selecting effective test data. Thus the problem has been converted from its usual form as an informal study of programs and programming to a more formal investigation of the geometry of input space domains.

The strategy is initially described for the case of linear predicates and a two-dimensional input space. For the linear case, it is shown that, under appropriate assumptions, the number of test points to reliably test a domain grows only linearly with the number of predicates along the path and with the dimensionality. The techniques are then extended to N dimensions, and various other extensions are considered, including nonlinear predicates.

A domain boundary error analysis is presented in Chapter 6, which is helpful in choosing the best locations for test points. The application of the domain strategy in discrete spaces is analyzed to study the effect of roundoff error in selecting test points.

In the concluding Chapter 7 a number of open questions generated by this investigation are presented, and the prospects for the practical application of the domain testing strategy are evaluated.

CHAPTER 2

BACKGROUND AND PRELIMINARIES

2.1 Programming Language Assumptions

In order to investigate domain errors, we need to consider the language in which programs will be written. The control structures should be simple and concise, and should resemble those available in most procedure-oriented languages. For simplicity we assume a single real-valued data type, and this is converted to integer values for use as DO-loop indices. Because this is a path-oriented approach, no extra control flow problems are introduced by block structure. Thus no provision is made for block structure, as it would only add extra bookkeeping to keep track of local variables and block invocation or exit.

A number of programming language features are assumed not to occur in the programs we are to analyze for domain errors. The first feature is that of arrays; despite the fact that arrays commonly occur in programs, a predicate which refers to an element of an input array can cause major complications (Ramamoorthy [11]). A second class of language features which will be excluded in our analysis is that of subroutines and functions. The problems of side effects and of parameter passing pose difficulties for domain testing. The third class of features which are not currently analyzed by domain testing include nonnumerical data types such as character data and pointers. These are admittedly very important features, and further research is needed to investigate whether these features pose any fundamental limitations to the domain testing strategy.

Since input/output processing is so closely linked to a machine or compiler environment, we will assume that all I/O errors have previously been eliminated. Thus only the most elementary I/O capabilities are provided; input is provided by a simple READ statement, and output is accomplished with a simple WRITE statement.

The types of control flow constructs investigated in this research include sequence, alternation, and iteration control. Since the analysis is path-oriented, GO-TO statements could be included without adversely affecting any results, except that program paths could become quite complex.

All computation is accomplished by means of arithmetic assignment statements which also provide the basic sequential flow of control. In each statement a single variable is assigned a value. The right hand side of an assignment statement is an arithmetic expression using variables, constants, and a set of basic arithmetic operators (+, -, *, /).

The general predicate form used for control flow is a Boolean combination of arithmetic relational expressions. The logical operators OR and AND are used to form these Boolean combinations. Each arithmetic relational expression contains a relational operator from the set (<, >, =, ≤, ≥, ≠). These operators form a complete set, and thus the logical operator NOT is unnecessary. If a predicate consists of two or more relational expressions with Boolean operators, then it is a compound predicate. A simple predicate consists of just a single relational expression.

The alternation type of control flow is achieved by using the IF-THEN-ELSE-ENDIF construct. The conditional associated with the IF statement is a general predicate. Any well-formed program segment, including the null program segment, can be used in the THEN and ELSE portions of the IF construct. The ENDIF statement is just a delimiter for the IF construct, which clarifies the nesting structure and eliminates any potentially ambiguous ELSE clause.

A general iteration construct is included which consists of a DO statement, loop body, and ENDDO delimiter. The DO statement can be in one of three forms:

1) DO I = INIT, FINAL, INCR;

2) DO WHILE (general predicate);

3) DO I = INIT, FINAL, INCR WHILE (general predicate).

The loop body can be any well-formed program segment, and the ENDDO is just a delimiter to clarify the scope of the iteration.

The variables used in a program are divided into three classes. If a variable appears in a READ or WRITE statement, it is classified as an input or output variable respectively; all other variables are called program variables. In order to produce a clear delineation between the three types of variables, we assume that a given variable belongs to only one of the above three classes.

2.2 Program Paths and Path Predicates

A program can be represented as a directed graph $G = (V,A)$, where V is a set of nodes and A is the set of arcs or directed edges between nodes. In the language discussed in Section 2.1, we have defined a set of basic program elements which consists of a READ, WRITE, assignment, IF, and DO statement, together with the ENDIF and ENDDO delimiters. The directed graph representation of a program will contain a node for each occurrence of a basic program element, and an arc for each possible flow of control between these elements. While THEN and ELSE statements do not explicitly appear in the digraph, the actions associated with them will be represented as nodes in the digraph.

A walk in a digraph is defined as an alternating sequence of nodes and arcs $(v_1, A_{12}, v_2, A_{23}, \ldots, A_{k-1,k}, v_k)$ such that each arc $A_{i,i+1}$ is directed from node v_i to node v_{i+1}. A control path is then defined to be a walk in the directed graph beginning with the node for the initial statement and terminating with the node for the final statement. It should be noted that two walks which differ only in the number of times a particular loop in the program is executed will be defined as two distinct control paths. Thus the number of control paths in a program can be infinite.

Every branch point of the program is associated with a general predicate. This predicate evaluates to true or false, and its value determines which outcome of the branch will be followed. A predicate is generated each time control reaches an IF or DO statement in the given language. The path condition is the compound condition which must be satisfied by the input data point in order for the control path to be executed. It is the conjunction of the individual predicate conditions which are generated at each branch point along the control path. Not all the control paths that exist syntactically within the program are executable. If input data exist which satisfy the path condition, the control path is also an <u>execution path</u> and can be used in testing the program. If the path condition is not satisfied by any input value, the path is said to be <u>infeasible</u>, and is of no use in testing the program.

A simple predicate is said to be <u>linear</u> in variables $V_1, V_2, \ldots\ldots, V_n$ if it is of the form

$$A_1V_1 + A_2V_2 + \ldots . + A_nV_n \quad \text{ROP} \quad K,$$

where K and the A_i are constants, and ROP represents one of the relational operators ($<,>,=,\leq,\geq,\neq$). A compound predicate is linear when each of its component simple predicates is linear.

In general, predicates can be expressed in terms of both program variables and input variables. However, in generating input data to satisfy the path condition we must work with constraints in terms of only input variables. If we replace each program variable appearing in the predicate by its symbolic value in terms of input variables, we get an equivalent constraint which we call the <u>predicate interpretation</u>. A particular interpretation is equivalent to the original predicate in that input variable values satisfying the interpretation will lead to the computation of program variables which also satisfy the original predicate. A single predicate can have many different interpretations depending upon which path is selected, for each path will in general consist of a different sequence of assignment statements. The following program segment provides example predicates and interpretations.

```
READ A,B;

IF A > B
   THEN C = B + 1;
   ELSE C = B - 1;
ENDIF;
D = 2*A + B;
IF C ≤ 0
  THEN E = 0;
  ELSE
       DO I = 1,B;
          E = E + 2*I;
       ENDDO;
ENDIF;
IF D = 2
   THEN F = E + A;
   ELSE F = E - A;
ENDIF;

WRITE F;
```

In the first predicate, $A > B$, both A and B are input variables, so there is only one interpretation. The second predicate, $C \leq 0$, will have two interpretations depending on which branch was taken in the first IF construct. For paths on which the THEN $C = B + 1$ clause is executed the interpretation is $B + 1 \leq 0$ or equivalently $B \leq -1$. When the ELSE $C = B - 1$ branch is taken, the interpretation is $B - 1 \leq 0$, or equivalently $B \leq 1$. Within the second IF-THEN-ELSE clause, a nested DO-loop appears. The DO-loop is executed:

no times if $B < 1$

once if $1 \leq B < 2$

twice if $2 \leq B < 3$

etc.

Thus the selection of a path will require a specification of the number of times that the DO-loop is executed, and a corresponding predicate is applied which selects those input points which will follow that particular path. Even though the third predicate, $D = 2$, appears on four different paths, it only has one interpretation, $2*A + B = 2$, since D is assigned the value $2*A + B$ in the same statement in each of the four paths.

2.3 Importance of Linear Predicates

The domain testing strategy becomes particularly attractive from a practical point of view if the predicates are assumed to be linear in input variables. It might seem to be an undue limitation to require that predicate interpretations be linear for the proposed strategy. In fact, however, as the following discussion shows, this represents no real limitation for many important applications.

A number of authors have provided data to show that simple programming language constructs are used more often than complex constructs. Knuth [10] studied a random sample of FORTRAN programs and found that 86% of all assignment statements were of the forms

$$V_1 = V_2,$$
$$V_1 = V_2 + V_3,$$
or $$V_1 = V_2 - V_3.$$

Also 70% of all DO loops in the programs contained less than four statements. Elshoff [5,6] studied 120 production PL/I programs and showed similar results, including the fact that 97% of all arithmetic operators are + or -, and 98% of all expressions contain fewer than two operators.

An experiment of particular relevance to the present context is reported in Cohen [4] using typical data processing programs, since program functions and programming practice tend to be reasonably uniform in this area. A random sample of 50 COBOL programs was taken directly from production data processing applications for this study. In this static analysis each predicate is classified according to whether it is linear or nonlinear, and the number of input variables used in the predicate has also been recorded. In addition, the number of input-independent predicates were tabulated, since these predicates do not produce any input constraints. The number of equality predicates is also reported since these predicates are very beneficial in reducing the number of test points required for a domain. These data are summarized in Table I.

	TOTAL	AVG.	RANGE
Total Lines	12,628	253	31-1,287
Procedure Division Lines	8,139	163	13-822
Total Predicates	1,225	25	0-115
Linear Predicates	1,070	21	0-104
Nonlinear Predicates	1	0.02	0-1
Input-Independent Predicates	154	3	0-28
Predicates with 1 Variable	945	19	0-97
Predicates with 2 Variables	125	2.5	0-20
Equality Predicates	779	15.5	0-76

TABLE I Predicate Statistics for 50 COBOL Programs

The most important result is that only one predicate out of the 1225 tabulated in the study can possibly be a nonlinear predicate. The predicates are also very simple since most of them refer to only one input variable, and no predicate in this sample uses more than two input variables.

In conclusion, while this study by no means represents an exhaustive survey, we believe the sample is large enough to indicate that nonlinear predicate interpretations are rarely encountered in data processing applications. It is clear that any testing strategy restricted to linear predicates is still viable in many areas of programming practice.

2.4 Input Space Structure

A program which has N input variables and produces M output variables computes a function which maps points in the N-dimensional input space to points in the M-dimensional output space. The input space is partitioned into a set of domains. Each domain corresponds to a particular executable path in the program and consists of the input data points which cause the path to be executed. More formally, an input space domain is defined as a set of input data points satisfying a path condition, consisting of a conjunction of predicates along the path. In this discussion, these predicates are assumed to be simple; compound predicates will be discussed later in Section 5.3.

We assume that the input space is bounded in each direction by the minimum and maximum values for the corresponding variable. These min-max constraints do not appear in the program but are automatically appended to each path condition. Since a single data type is used for all variables in our language, each variable will have the same min-max constraints.

The boundary of each domain is determined by the predicates in the path condition and consists of border segments, where each segment is the section of the boundary determined by a single simple predicate in the path condition. Each border segment can be open or closed depending on the relational operator in the predicate. A closed border segment is actually part of the domain and is formed by predicates with $\leq$, $\geq$, or = operators. An open border segment forms part of the domain boundary but does not constitute part of the domain, and is formed by <, >, and $\neq$ predicates. We shall find it convenient to use the term border operator to refer to the relational operator for the corresponding predicate.

Since border segments in the input space are determined by the particular predicate interpretations on the path, the form of the segment may be different from that of the original predicate. For example, with input variables A and B, the linear predicate $A < C + 2$ can lead to a nonlinear border segment, $A < B*B + 2$, when $C = B*B$. Similarly, a nonlinear predicate, $C > A*A + B$, will produce a linear border segment, $A \geq B$, when $C = A*A + A$. Since a predicate can appear on many paths and each path can execute a different sequence of assignment statements for the variables used in the predicate, a single predicate can have many different interpretations and can form many discontinuous border segments for various domains.

The total number of predicates on the path is only an upper bound on the number of border segments in the domain boundary since certain predicates in the path condition may not actually produce border segments. An input-independent predicate interpretation is one which reduces to a relation between constants, and since it is either true or false regardless of the input values, it does not further constrain the domain. A redundant predicate interpretation is one which is superceded by the other predicate interpretations, i.e., the domain can be defined by a strict subset of the predicate interpretations for that path.

The general form of a simple linear predicate interpretation is

$$A_1 X_1 + A_2 X_2 + \dots + A_n X_n \quad \text{ROP} \quad K$$

where ROP is the relational operator, X_i are input variables, and A_i, K are constants. However, the border segment which any of these predicates defines is a section of the surface defined by the equality

$$A_1 X_1 + A_2 X_2 + \ldots + A_n X_n = K,$$

since this is the limiting condition for the points satisfying the predicate. In an N-dimensional space this linear equality defines a hyperplane which is the N-dimensional generalization of a plane.

Consider a path condition composed of a conjunction of simple predicates. These predicates can be of three basic types: equalities (=), inequalities (<, >, $\leq$, $\geq$), and nonequalities ($\neq$). The use of each of the three types results in a markedly different effect on the domain boundary. Each equality constrains the domain to lie in a particular hyperplane, thus reducing the dimensionality of the domain by one. The set of inequality constraints then defines a region within the lower dimensional space defined by the equality predicates.

The nonequality linear constraints define hyperplanes which are not part of the domain, giving rise to open border segments as mentioned earlier. Observe that the constraint $A \neq B$ is equivalent to the compound predicate $(A < B)$ OR $(A > B)$. In this form it is clear that the addition of a nonequality predicate to a set of inequalities can split the domain defined by those inequalities into two regions.

The following example should clarify the concepts discussed above,

```
        READ I,J;
        C = I + 2*J - 1;
(P1)    IF C > 6
          THEN D = C - I;
          ELSE D = C + I - J + 2;
        ENDIF;
(P2)    IF D = C + 2
          THEN E = I;
          ELSE E = 3;
        ENDIF;
(P3)    IF E <= D - 2*J
          THEN F = I;
          ELSE F = J;
        ENDIF;
        WRITE F;
```

Figure 1 shows the corresponding input space partitioning structure for this program. The input space is in terms of inputs I and J, and is arbitrarily constrained by the following min-max conditions:

$$-3 \leq I \leq 4, \qquad -2 \leq J \leq 6.$$

Each border in Figure 1 is labelled with the corresponding predicate, and each domain is labelled with the corresponding path. The path notation is based upon which branch (T or E) is taken in each of the three IF constructs, e.g., TEE.

The first predicate P1, C > 6, will be interpreted as I + 2*J > 7 since C = I + 2*J - 1. This single interpretation P1 is seen in Figure 1 as a single continuous border segment across the entire input space. The second predicate P2 demonstrates the effects of both equality and nonequality predicates. Domains for paths through the THEN branch are constrained by the equality, and this reduction in dimensionality is seen in the fact that these domains consist of the points on the solid line segments ETT and TTT. Paths through the ELSE branch are constrained by a nonequality predicate, and the corresponding domains consist of the two regions on either side of the solid line segments (e.g., EEE). This predicate has two interpretations depending upon the value assigned to D and produces two discontinuous border segments ETT and TTT.

The third predicate P3 might have four different interpretations, but only one border segment appears in the diagram. The other three interpretations do not produce borders since they are either redundant, input-independent, or correspond to infeasible paths. With three IF constructs we have eight control paths, but the diagram contains only five domains since three of the paths are infeasible. Also many of these domains have fewer than three border segments because of redundant and input-independent interpretations. From this example we can conclude that the input space partitioning structure of a program with many predicates and a larger dimensional input space can be extremely complicated.

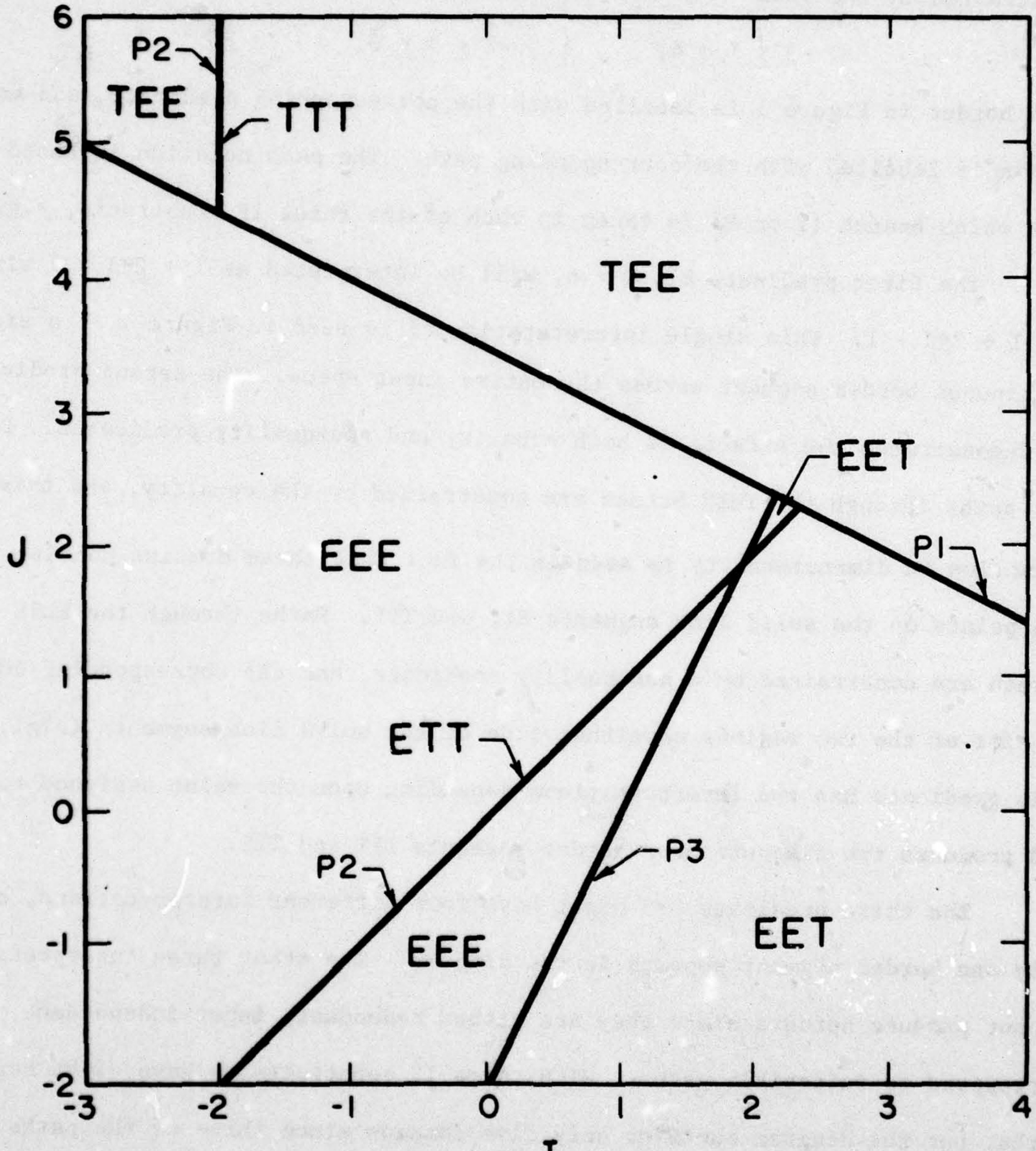


FIGURE 1 Input Space Partitioning Structure

The foregoing definitions and the example allow us to characterize more precisely domains which correspond to simple linear predicate interpretations. For a formal statement of the characterization, we need the following definitions. A set is convex, if for any two points in the set, the line segment joining these points is also in the set. A convex polyhedron is the set produced by the intersection of the set of points satisfying a finite number of linear equalities and inequalities.

Proposition 1

For an execution path with a set of simple linear equality or inequality predicate interpretations, the input space domain is a single convex polyhedron. If one or more simple linear nonequality predicate interpretations are added to this set, then the input space domain consists of the union of a set of disjoint convex polyhedra.

CHAPTER 3

ERROR CLASSIFICATION AND THEORETICAL LIMITATIONS

3.1 Definitions of Types of Error

The basic ideas behind the classification of errors that we use are due to Howden [9], but our approach to defining them is somewhat more operational than that given in his paper.

From the previous sections, it is clear that a program can be viewed as

1) establishing an exhaustive partition of the input space into mutually exclusive domains each of which corresponds to an executable path, and
2) specifying, for each domain, a set of assignment statements which constitute the domain computation.

Thus we have a canonical representation of a program, which is a (possibly infinite) set of pairs $\{(D_1;f_1),(D_2;f_2), \ldots (D_i;f_i),\ldots\}$, where D_i *is the i-th* domain, and f_i is the corresponding domain computation function.

Given an incorrect program P, let us consider the changes in its canonical representation as a result of modifications performed on P. It is assumed that these modifications are made using only permissible language constructs and results in a legal program.

Definition: A domain boundary modification occurs if the modification results in a change in the D_i component of some $(D_i;f_i)$ pair in the canonical representation.

Definition: A domain computation modification occurs if the modification results in a change in the f_i component of some $(D_i;f_i)$ pair in the canonical representation.

Definition: A <u>missing path modification</u> occurs if the modification results in the creation of a new $(D_i;f_i)$ pair such that D_i is a subset of D_j occurring in some pair $(D_j;f_j)$ in the canonical representation of P, and f_j differs from f_i.

Notice that a particular modification (say a change of some assignment statement) can be a modification of more than one type. In particular, a missing path modification is also a domain boundary modification.

The errors that occur in a program can be classified on the basis of the modifications needed to obtain a correct program and consequent changes in the canonical representation. In general, there will be many correct programs, and multiple ways to get a particular correct program. Hence, the error classification is not unique, but relative to the particular correct program that would result from the series of modifications.

<u>Definition</u>: An incorrect program P can be viewed as having a <u>domain error</u> (<u>computational error</u>) (<u>missing path error</u>) if a correct program P^* can be created by a sequence of modifications at least one of which is a domain boundary modification (domain computation modification)(missing path modification).

Several remarks are in order. The operational consequence of the phrase "can be viewed as" in the above definition is that the error classification is relative not only to a particular correct program, but also to a particular sequence of modifications. For instance, consider an error in a predicate interpretation such that an incorrect relational operator is employed, e.g., use of > instead of <. This could be viewed as a domain error, leading to a modification of the predicate, or as a computation error, leading to a modification of the functions computed on the two branches. The fact that it might be more profitable to change the relational operator rather than the function computations is a consequence of the language constructs, and is not directly captured in the definitions of the types of error. In this paper we would regard an error due to an incorrect relational operator as a domain error; it is a simpler modification to change the relational operator in the predicate than to interchange the set of assignment statements.

More specific characterizations of these errors can be made in the context of the specific programming language which we have introduced. In particular, the following informal description directly relates the domain and missing path errors to the predicate constructs allowed in the language.

A path contains a domain error if an error in some predicate interpretation causes a border segment to be "shifted" from its correct position or to have an incorrect border operator. A domain error can be caused by an incorrectly specified predicate or by an incorrect assignment statement which affects a variable used in the predicate. An incorrect predicate or assignment statement may affect many predicate interpretations and consequently cause more than one border to be in error.

A path contains a missing path error when a predicate is missing which would subdivide the domain and create a new execution path for one of the subdomains. This type of error occurs when some special condition requiring different processing is omitted.

3.2 Fundamental Limitations

Finite testing strategies are fundamentally limited by their inability to detect phenomena occuring in regions which have zero volume or measure relative to the input space or domain. The first of these limitations we shall define as coincidental correctness. In testing each domain for the correctness of its boundaries, if the output for a test case is correct, it could be either that the test point was in the correct domain, or that it was in a wrong domain but the computation in that domain coincidentally yielded a correct value for the test point. Similarly, a domain computation could correspond to an incorrect function, but its output may coincide with the correct value for a particular test point. To be absolutely certain that the values are not coincidentally correct, it would be necessary to exhaustively test all the points of the domain.

The essence of the coincidental correctness problem is the same as that of the problem of deciding if two arbitrary computations are equivalent; the latter problem is known to be generally undecidable. However, in practice, the severity of the problem is related to the probability that for an arbitrary point this coincidence would occur. If the set of points for which the two functions have the same value is of measure zero, then this probability is zero, even though coincidental correctness is still possible. So, even with coincidental correctness as a possibility, a testing strategy can be _almost reliable_ in the sense of Howden [9], if it would be reliable in the absence of coincidental correctness, and the set of points which are coincidentally correct has zero volume relative to the domain being tested.

Another basic limitation relates to missing path errors. When the subdomain associated with a missing path is a region of lower dimensionality than the original domain, a _missing path error of reduced dimensionality_ occurs. This typically happens when the missing predicate is an equality. If all that is available is just the (incorrect) program to be tested, then the probability that a finite set of test points would detect the missing predicate is zero, since the volume of the subdomain is zero relative to that of the original domain.

The proposed approach is capable of detecting many kinds of missing path errors, but for some of them the number of required test points is inordinate. Hence, in the next section, where we describe the testing strategy, we will simply assume that no missing path errors are associated with the path being tested.

THE DOMAIN TESTING STRATEGY

The domain testing strategy is designed to detect domain errors and will be effective in detecting errors in any type of domain border under certain conditions. Test points are generated for each border segment which, if processed correctly, determine that both the relational operator and the position of the border are correct. An error in the border operator occurs when an incorrect relational operator is used in the corresponding predicate, and an error in the position of the border occurs when one or more incorrect coefficients are computed for the particular predicate interpretation. The strategy is based on a geometrical analysis of the domain boundary and takes advantage of the fact that points on or near the border are most sensitive to domain errors. A number of authors have made this observation, e.g., Boyer et al. [1] and Clarke [2].

As stated in Proposition 1, a domain defined by simple linear predicates is a convex polyhedron, and each point can be classified according to its position within the domain. An <u>interior point</u> is defined as one which is surrounded by an ε-neighborhood containing only points in the domain. Similarly, a <u>boundary point</u> is one for which every ε-neighborhood contains both points in the domain and points lying outside of the domain. Finally, an <u>extreme point</u> is a boundary point which does not lie between any two distinct points in the domain.

In the previous section, a comparison was made between the given program and a corresponding correct program; indeed domain errors were defined in terms of this correspondence. It should be emphasized that the domain strategy does not require that the correct program be given for the selection of test points, since only information obtained from the given program is needed. However, it will be convenient to be able to refer to a "correct border", although it will not be necessary to have any knowledge about this border. Define the given border as that corresponding to the predicate interpretation for the given program being tested, and the correct border as that border which would be calculated in some correct program.

The domain testing strategy is first developed, explained, and validated in detail under a set of simplifying assumptions:

1) Coincidental correctness does not occur for any test case. If correct output results are produced, we can assume that the test point is in the correct domain rather than being coincidentally correct in another domain.
2) A missing path error is not associated with the path being tested. Missing path errors of reduced dimensionality pose a theoretical limitation to the reliability of any program testing methodology.
3) Each border is produced by a simple predicate.
4) The path corresponding to each adjacent domain computes a different function than the path being tested.
5) The given border is linear, and if it is incorrect, the correct border is also linear.
6) The input space is continuous rather than discrete.
7) Each border is produced by an inequality predicate.
8) The input space is two-dimensional, corresponding to a program which reads at most two input variables.

The first two assumptions were thoroughly explored in the previous section.

Assumptions 3) through 8) are for convenience in the initial exposition, and we shall investigate later the conditions under which each can be relaxed. Also, references [3] and [4] discuss both the domain strategy and these assumptions in greater detail.

4.1 The Two-Dimensional Linear Case

Given assumptions 1) - 8), a set of test points is first defined for detecting border shifts, and then we shall show that this set of points also detects all possible relational operator errors. Since the present analysis is limited to linear borders in a two-dimensional input space, each border is a line segment. Therefore, the correct border can be determined if we know two points on that border.

The test cases selected will be of two types, defined by their position with respect to the given border. An ON test point lies on the given border, while an OFF test point is a small distance ε from, and lies on the open side of, the given border. Therefore, we observe that when testing a closed border, the ON test points are in the domain being tested, and each OFF test point is in some adjacent domain. Conversely, when testing an open border, each ON test point is in some adjacent domain, while the OFF test points are in the domain being tested.

Figure 2 shows the selection of three test points A, B, and C for a closed inequality border segment. In this and subsequent figures the small arrows are used to indicate the domain which contains the border segment. The three points must be selected in an ON-OFF-ON sequence. Specifically, if test point C is projected down on line AB, then the projected point must lie strictly between A and B on this line segment. Also point C is selected a distance ε from the given border segment, and will be chosen so that it satisfies all the inequalities defining domain D except for the inequality being tested.

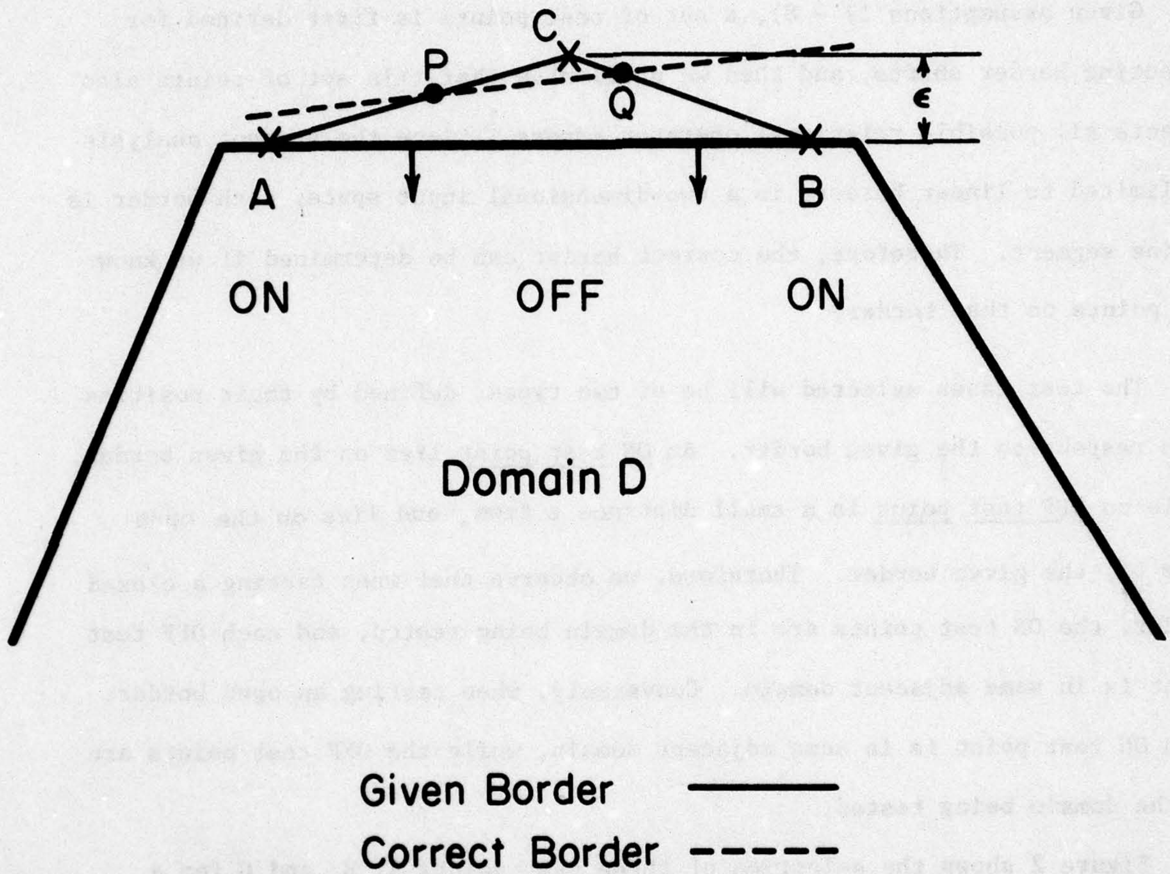


FIGURE 2 Test Points for a Two-Dimensional Linear Border

It must be shown that test points selected in this way will reliably detect domain errors due to boundary shifts. If any of the test points lead to an incorrect output, then clearly there is an error. On the other hand, if the outputs of all these points are correct, then either the given border is correct or we have gained considerable information as to the location of a correct border. Figure 2 shows that the correct border must lie on or above points A and B, and must lie below point C, for by assumptions (1) and (4), each of these test points must lie in its assumed domain. So if the given border is incorrect, the correct border can only belong to a class of line segments which intersect both closed line segments AC and BC.

Figure 2 indicates a specific correct border from this class which intersects line segments AC and BC at P and Q respectively. Define the domain error magnitude for this correct border to be the maximum of the distances from P and from Q to the given border. Then it is clear that the chosen test points have detected domain errors due to border shifts except for a class of domain errors of magnitude less than ε. In a continuous space ε can be chosen arbitrarily small, and as ε approaches zero, the line segments AC and BC become arbitrarily close to the given border, and in the limit, we can conclude that the given border is identical to the correct border. However, the continuity of the space also implies that regardless of how small ε is chosen, border shifts of magnitude less than ε may not be detected, and therefore we must correspondingly qualify our results.

Figure 3 shows the three general types of border shifts, and will allow us to see how the ON-OFF-ON sequence of test points works in each case. In Figure 3(a), the border shift has effectively reduced domain D_1. Test points A and B yield correct outputs, for they remain in the correct domain D_1 despite the shifted border. However, the border has shifted past

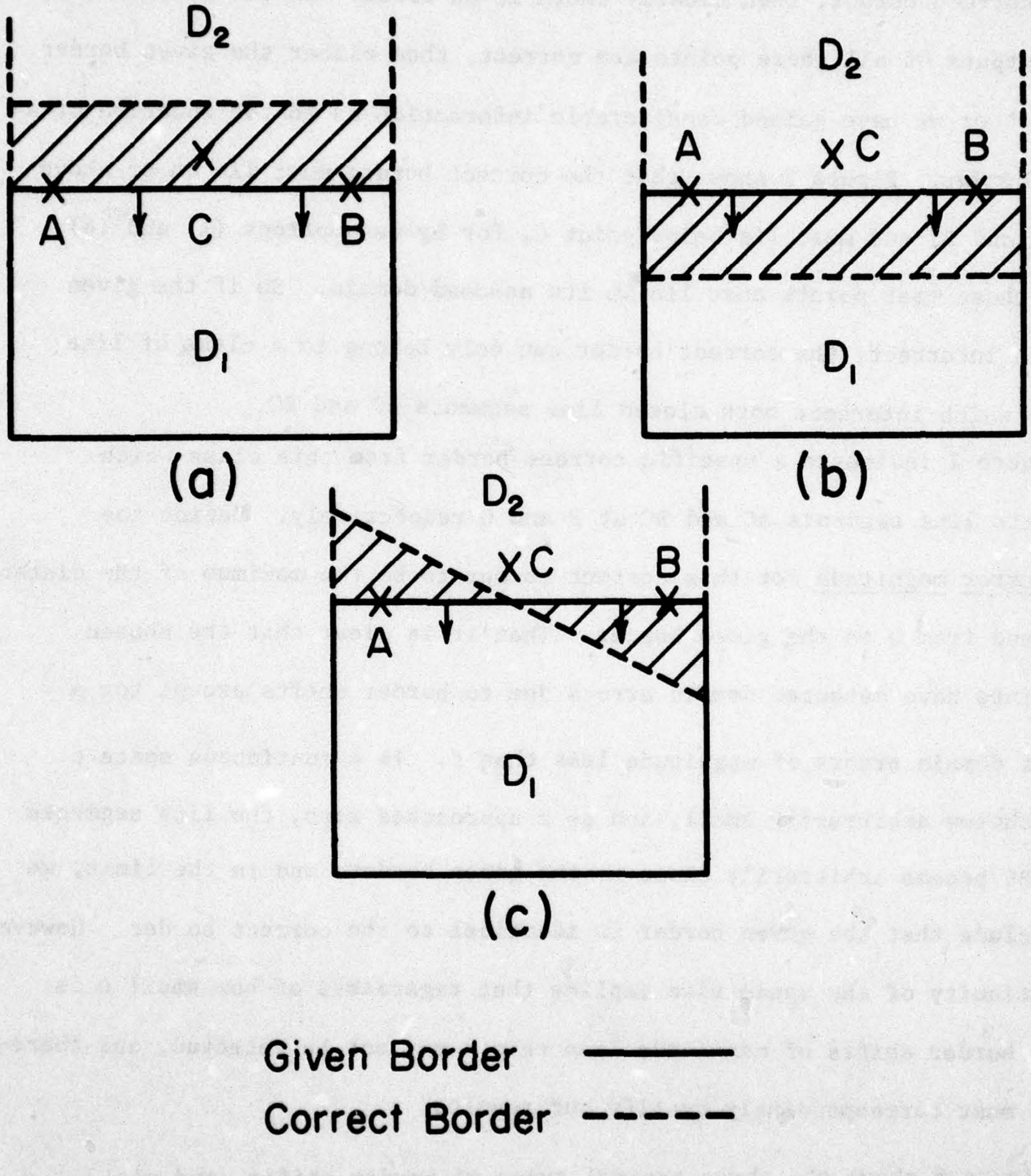


FIGURE 3 The Three Types of Border Shifts

test point C, causing it to be in domain D_2 instead of domain D_1. Since the program will now follow the wrong path when executing input C, incorrect results will be produced. In Figure 3(b), the domain D_1 has been enlarged due to the border shift. Here test point C will be processed correctly since it is still in domain D_2, but both A and B will detect the shift since they should also be in domain D_2. Finally in Figure 3(c), only test point B will be incorrect since the border shift causes it to be in D_1 instead of D_2. Therefore, the ON-OFF-ON sequence is effective since at least one of the three points must be in the wrong domain as long as the border shift is of a magnitude greater than ε.

Recall in Figure 2 that we required the OFF point C to satisfy all the inequalities defining domain D except for the inequality being tested. The reason for this requirement is that some correct border segment may terminate on the extension of an adjacent border, rather than intersecting both line segements AC and BC as we have argued. Since we have assumed a continuous space, C could always be chosen closer to the given border in order to satisfy the adjacent border inequalities. An analysis of this situation will be presented in Section 6.2.

We must also demonstrate the reliability of the method for domain errors in which the predicate operator is incorrect. If the direction of the inequality is wrong, e.g., $\leq$ is used instead of $\geq$, the domains on either side of the border are interchanged, and any point in either domain will detect the error. A more subtle error occurs when just the border itself is in the wrong domain, e.g., $\leq$ is used instead of $<$. In this case the only points affected lie on the border, and since we always test ON points, this type of error will always be detected. If the correct predicate is an equality, the OFF point will detect the error.

The domain testing strategy requires at most 3*P test points for a domain, where P, the number of border segments on this boundary, is bounded by the number of predicates encountered on the path. However, we can reduce this cost by sharing test points between adjacent borders of the domain. The requirement for sharing an ON point is that it is an extreme point for two adjacent borders which are both closed or both open. In the example in Figure 4, the points that can be shared are A_1, A_2, and A_3. The number of ON points needed to test the entire domain boundary can be reduced by as much as one half, i.e., the number of test points, TP, required to test the complete domain boundary lies in the following range:

$$2*P \leq TP \leq 3*P.$$

Even more significant savings are possible by sharing the test points for a common border between two adjacent domains. If both domains are tested independently, the common border between them is tested twice, using a total of six test points. If this border has shifted, both domains must be affected, and the error will be detected by testing either domain. Therefore, the second set of test points can safely be omitted. However, the cost savings in such sharing should be balanced against the additional processing required.

We now formally summarize the results of this section in the following proposition.

<u>Proposition 2</u>

Given assumptions (1) through (8), with the OFF test point chosen a distance ϵ from the corresponding border, the domain testing strategy is guaranteed to detect all domain errors of magnitude greater than ϵ. Moreover, the cost is no more than 3*P test points per domain, where P is the number of predicates along the corresponding path.

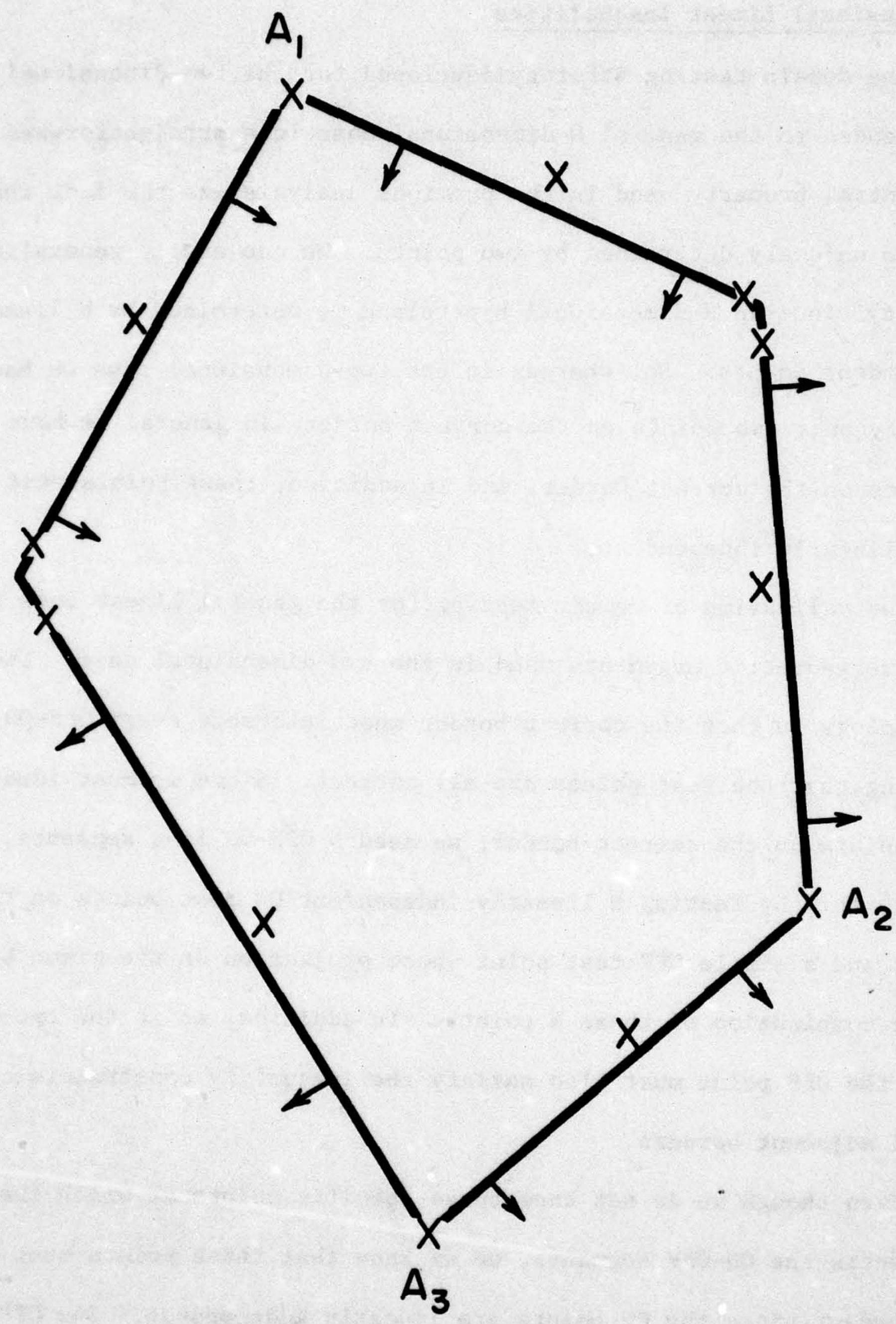


FIGURE 4 Domain Test Points for Closed and Open Borders

4.2 N-Dimensional Linear Inequalities

The domain testing strategy developed for the two-dimensional case can be extended to the general N-dimensional case in a straightforward manner. The central property used in the previous analysis was the fact that a line is uniquely determined by two points. We can easily generalize this property since an N-dimensional hyperplane is determined by N linearly independent points. So, whereas in the two-dimensional case we had to identify only two points on the correct border, in general we have to identify N points on the correct border, and in addition, these points must be guaranteed to be linearly independent.

The validation of domain testing for the general linear case is based on the same geometric arguments used in the two-dimensional case. The key to the methodology is that the correct border must intersect every OFF-ON line segment, assuming that the test points are all correct. Since we must identify a total of N points on the correct border, we need N OFF-ON line segments, and we can achieve this by testing N linearly independent ON test points on the given border and a single OFF test point whose projection on the given border is a convex combination of these N points. In addition, as in the two-dimensional case, the OFF point must also satisfy the inequality constraints corresponding to all adjacent borders.

Even though we do not know these specific points at which the correct border intersects the ON-OFF segments, we do know that these points must be linearly independent since the ON points are linearly independent. The OFF point is a distance ε from the given border, and in the limit as ε approaches zero, each OFF-ON line segment becomes arbitrarily close to the given border. However, as in the two-dimensional case, the ε-limitation means that only border shifts of magnitude greater than ε will be detected.

The domain testing strategy requires at most (N+1)*P test points per domain, where N is the dimensionality of the input space in which the domain is defined and P is the number of border segments in the boundary of the specific domain. However, we again can reduce this testing cost by using extreme points as ON test points. Each extreme point is formed by the intersection of at least N border segments, and therefore one point can be used to test up to N borders. In addition, extreme points are also linearly independent. Each border must be tested by N ON points, and any points beyond this are redundant, and so not all extreme points on each border are required. As a result of this kind of sharing, the number of test points can be as few as 2*P. As in the two-dimensional case, there can be further savings if test points are shared between adjacent domains. Finally, since some of the P border segments may be produced by the min-max constraints which define the bounds of the input space, the number of test points can be reduced still further, if we can assume that these constraints are predetermined and need not be tested.

This generalization to N dimensions is significant since very few nontrivial programs have only two input variables. We summarize the results so far in the following proposition:

<u>Proposition 3</u>

Given assumptions (1) - (7), with the OFF test point chosen a distance ε from the corresponding border, the domain testing strategy is guaranteed to detect all domain errors of magnitude greater than ε regardless of the dimensionality of the input space. Moreover, the cost is not more than (N+1)*P test points per domain.

4.3 Equality and Nonequality Predicates

Equality predicates constrain the domain to lie in a lower dimensional space. If we have an N-dimensional input space and the domain is constrained by L independent equalities, the remaining inequality and nonequality predicates then define the domain within the (N-L)-dimensional subspace defined by the set of equality predicates.

In Figure 5 we see the equality border and the proposed set of test points. In a general N-dimensional domain, let us first consider a total of N ON points on the border and two OFF points, one on either side of the border. As before, the ON points must be independent, and the projection of each OFF point on the border must be a convex combination of the ON points.

Given an incorrect equality predicate, the error could be either in the relational operator or in the position of the border or both. The proposed set of test points can be shown to detect an operator error or a position error by arguments analogous to those previously given. This set of points is also adequate for almost all combinations of operator and position errors, except for the following pathological possibility. Let us assume that the border has shifted and the correct predicate is a nonequality. If both OFF points happen to lie on the correct border while none of the ON points belong to this border, the error would go undetected. This singular situation is diagrammed as the dashed border in Figure 6, where A_1 and A_2 are the ON points, and C_1 and C_2 are the OFF points. This problem can be solved by testing one additional point selected so that it lies both on the given border and the correct border for this case, i.e., at the intersection point of the given border with the line segment connecting the two OFF points. This additional point is denoted by B in the figure.

Each equality predicate can thus be completely tested using a total of (N+3) test points. By sharing test points between all the equality predicates,

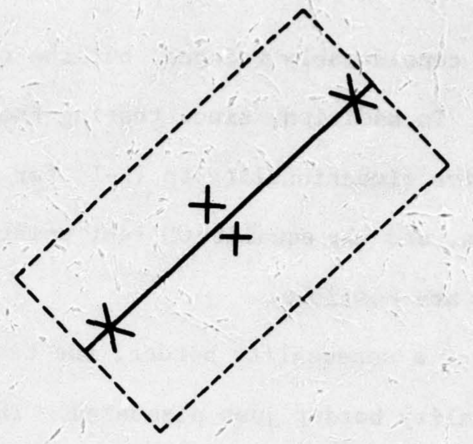


FIGURE 5 Test Points for an Equality Border

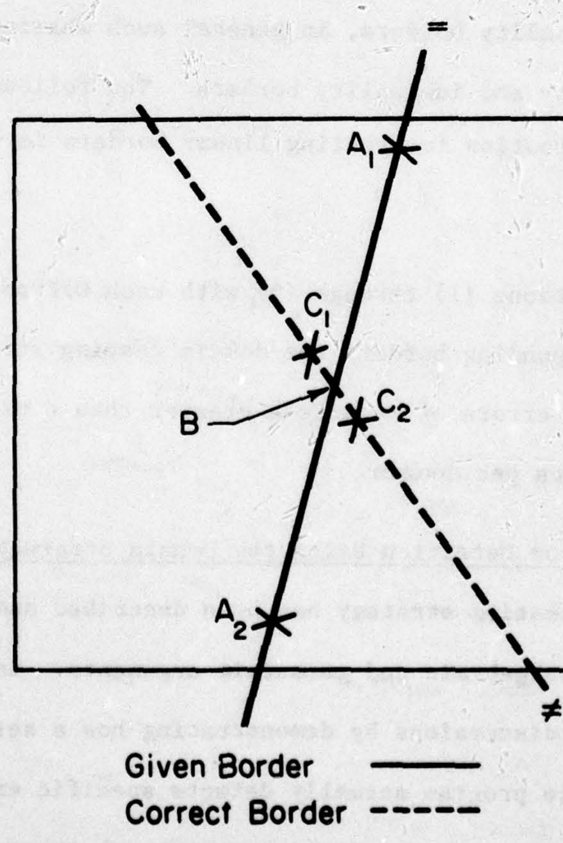


FIGURE 6 A Pathological Case in Domain Testing for an Equality Predicate

this number can be considerably reduced, but the reduction depends upon values of N and L. In addition, since testing the equality predicates reduces the effective dimensionality to (N-L) for each of the inequality and nonequality borders, and the equality ON test points can be shared, even further reductions are possible.

For the case of a nonequality border, the testing strategy is identical to that of the equality border just discussed. The arguments for the validity of the strategy are analogous to those in previous cases. Again in this case, the pathological possibility discussed in connection with the equality predicate can occur, and can be handled in the same way. The major difference is that while test points can be extensively shared between equality and inequality borders, in general such sharing is not possible between nonequality and inequality borders. The following proposition summarizes the situation for testing linear borders in N-dimensions.

Proposition 4

Given assumptions (1) through (6), with each OFF point chosen a distance ϵ from the corresponding border, the domain testing strategy is guaranteed to detect all domain errors of magnitude greater than ϵ using no more than P*(N+3) test points per domain.

4.4 An Example of Error Detection Using the Domain Strategy

The domain testing strategy has been described and validated using somewhat complicated algebraic and geometric arguments. In this section we hope to complement those discussions by demonstrating how a set of domain test points for a short sample program actually detects specific examples of different types of programming errors. In discussing each error we will focus on a specific domain affected by the error, and a careful analysis of its effect on the domain will allow us to identify those domain test points which detect the error.

The short example program reads two values, I and J, and produces a single output value M. Therefore, the input space is two-dimensional, and the following min-max constraints have been chosen so that the input space diagram would not be too large or complicated.

$$-8 \leq I \leq 8 \qquad -5 \leq J \leq 5.$$

In addition, since this is a two-dimensional space, we will also test extreme points for the border segments produced by the min-max constraints in order to be able to detect as many missing path errors as possible.

Even though the input space is assumed to be continuous, the coordinates of each test point are specified to an accuracy of 0.2 in order to simplify the diagrams and discussions. Of course, in an actual implementation each OFF point would be chosen much closer to the border.

The sample program is listed below, and it consists of three simple IF constructs, the first two of which are inequalities and the last of which is an equality. The input space structure is diagrammed in Figure 7, where the solid diagonal border across the entire space is produced by the first predicate, the dashed horizontal border and short vertical border at I=0 are produced by the second predicate, and the vertical equality border at I=5 corresponds to the third predicate. In addition, domain test points have been indicated for the two domains which we will discuss, viz., TTE and ETT.

```
Statement
Number

          READ I,J;

1         IF I ≤ J + 1
2           THEN K = I + J - 1;
3           ELSE K = 2*I + 1;
          ENDIF;

4         IF K ≥ I + 1
5          THEN L = I + 1;
6          ELSE L = J - 1;
          ENDIF;

7         IF I = 5
8           THEN M = 2*L + K;
9           ELSE M = L + 2*K - 1;
          ENDIF;

          WRITE M;
```

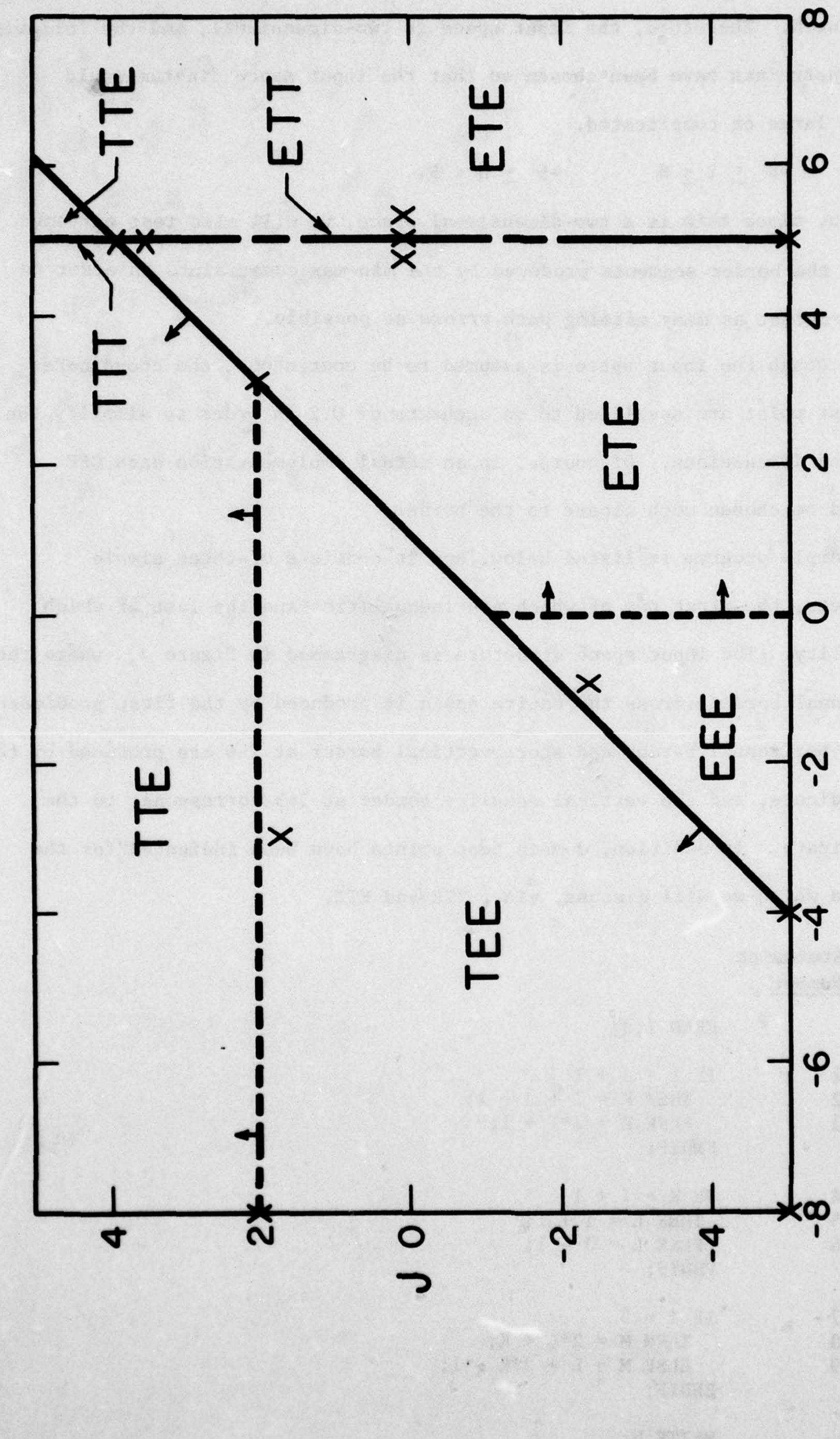


FIGURE 7 Input Space Domain Test Points

Table II illustrates two types of errors we would like to consider. The first is an error in the inequality predicate in statement #4 of the above program, ($K \geq I+1$), where it is assumed that the correct predicate should be ($K \geq I+2$). This corresponds to an inequality border shift, and the modified domain structure is shown in Figure 8. Three points have been selected to test this border, and it can be seen in Table II that the two ON points detect this error, where M and M' represent the output variables for the given program and for the assumed correct program respectively. Note that as a result of this error, the vertical border at I=0 in Figure 7 has also shifted to I=1 in Figure 8, and if tested, would also reveal this error.

Table II also shows the effect of an error in an equality predicate in statement #7 of the given program. It is assumed that the correct predicate should be (I=5-J) rather than the (I=5) predicate which occurs in the given program. Figure 9 shows the modified input space structure, and it can be seen that equality borders TTT and ETT have shifted. Table II shows the five points which test the ETT border, and note that two ON points both detect this shift.

Table III indicates that the domain strategy can also detect a computation error and a missing path error, even though we have previously noted that reliability cannot be proven for these cases. The computation error arises from statement #6 in the given program, where it is assumed that the correct assignment statement for this ELSE clause is (L=I-2) instead of (L=J-1) which actually appears in the given program. Since L is not used in any subsequent predicate, this corresponds to a computation error rather than a domain error. Thus the input space structure in Figure 7 is applicable for both the given and the correct programs. Table III shows the six test points which have been chosen to test domain TEE which is affected by this computation error. Four of the points should indicate the error, but note the test results at (-4, -5) are coincidentally correct; the remaining three points detect the error.

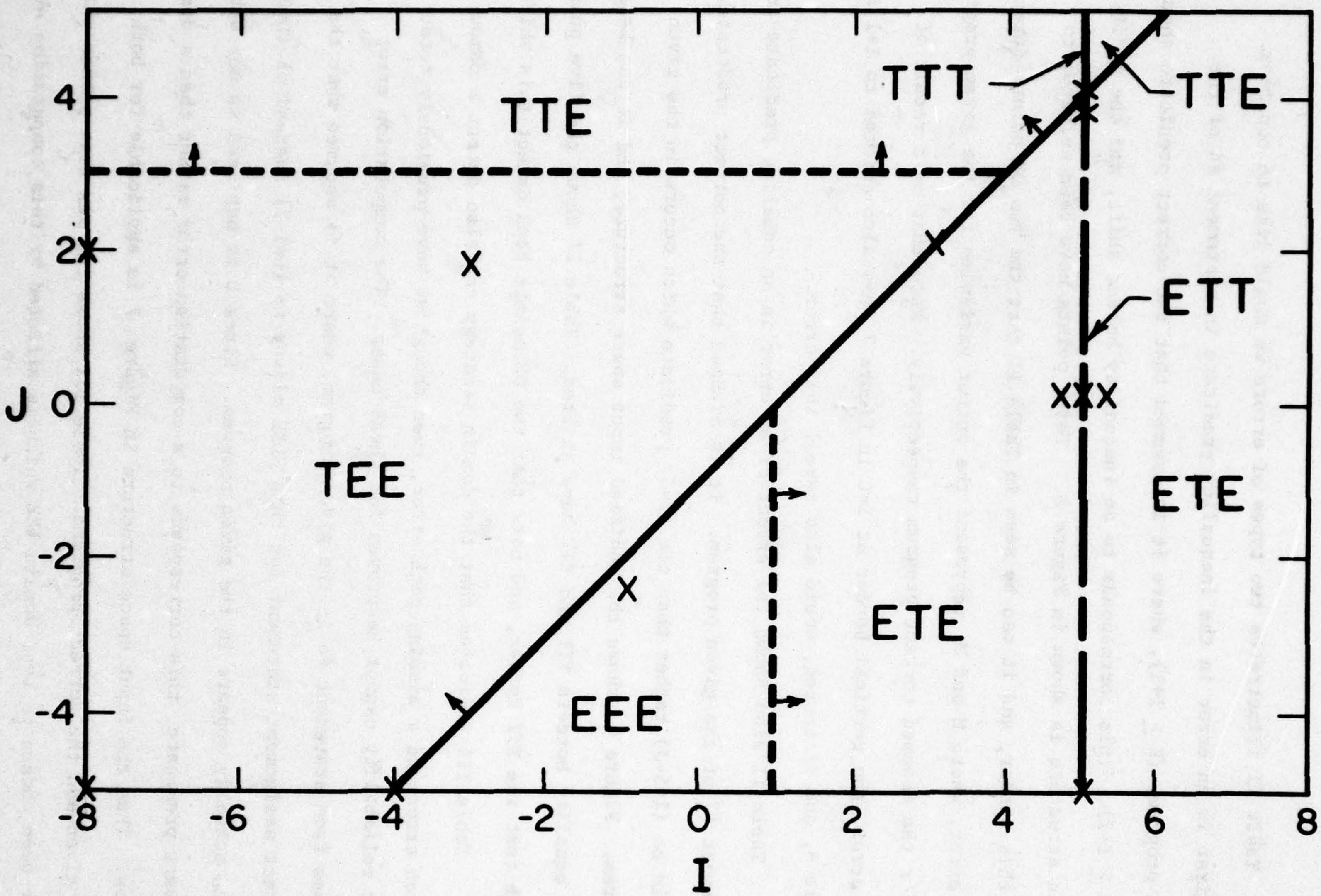


FIGURE 8 Correct Input Space for a Domain Error

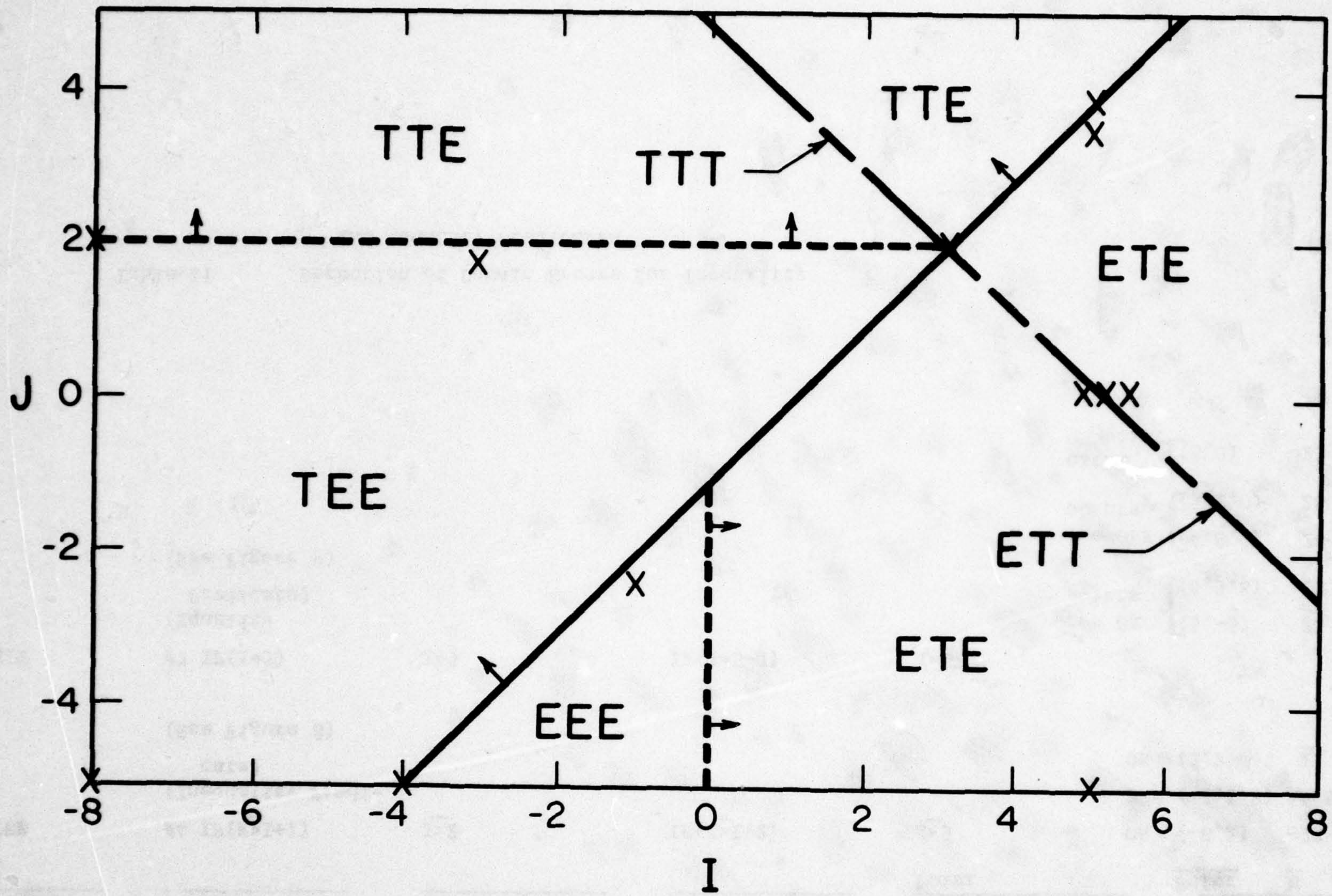


FIGURE 9 Correct Input Space for an Equality Predicate Error

Domain in Error	Given Statement in Error	Given Predicate Interpretation	Assumed Correct Statement	Correct Predicate Interpretation Point		Test Points for this Border Point	M	M'
TEE	#4 IF($K \geq I+1$)	$J \geq 2$	IF($K \geq I+2$)	$J \geq 3$	ON	(-8,2)	-22	14
	(Inequality Predicate)				OFF	(-3,1.8)	-4.6	-4.6
					ON	(3,2)	11	8
	(See Figure 8)							
ETT	#7 IF(I=5)	I=5	IF(I=5-J)	I=5-J				
	(Equality Predicate)				two ON points	(5,-5)	23	27
	(See Figure 9)					(5,3.8)	23	27
					two OFF points	(4.8,0)	26	26
						(5.2,0)	28	28
					extra ON point	(5,0)	23	23

Table II Detection of Domain Errors for Inequality and Equality Predicates

Domain in Error	Given Statement in Error	Assumed Correct Statement	Point	Test Points for this Domain M	M'
TEE (Computation Error) (See Figure 7)	#6 ELSE(L=J-1);	ELSE (L=I-2);	(-8,-5)	-35	-39
			*(-4,-5)	-27	-27
			(-3,1.8)	-4.6	-10.4
			(-1,-2.2)	-6.2	-6
			(-8,2)	-21	-21
			(3,2)	12	12
TEE (Missing Path Error) (See Figure 10)	#2 THEN(K=I+J-1);	THEN IF(2*J<-5*I -40) THEN K=3; ELSE K=I+J-1; ENDIF;	(-8,-5)	-35	-1
			(-4,-5)	-27	-27
			(-3,1.8)	-4.6	-4.6
			(-1,-2.2)	-6.2	-6.2
			(-8,2)	-21	-21
			(3,2)	12	12

* Note this point is coincidentally correct.

Table III Detection of a Computation Error and Missing Path Error

Suppose in program statement #2 the THEN clause is replaced by the following code.

```
THEN IF 2*J < -5*I - 40
     THEN K = 3;
     ELSE K = I + J - 1;
   ENDIF;
```

This corresponds to a missing path error and is indicated as such in Table III. Figure 10 shows how the domain TEE is modified by this missing path error, but note that only test point (-8,-5) detects this error. If the < inequality in the missing predicate had been an equality, this would have produced a missing path error of reduced dimensionality, corresponding to a domain consisting of just the line segment in Figure 10, and would have gone undetected.

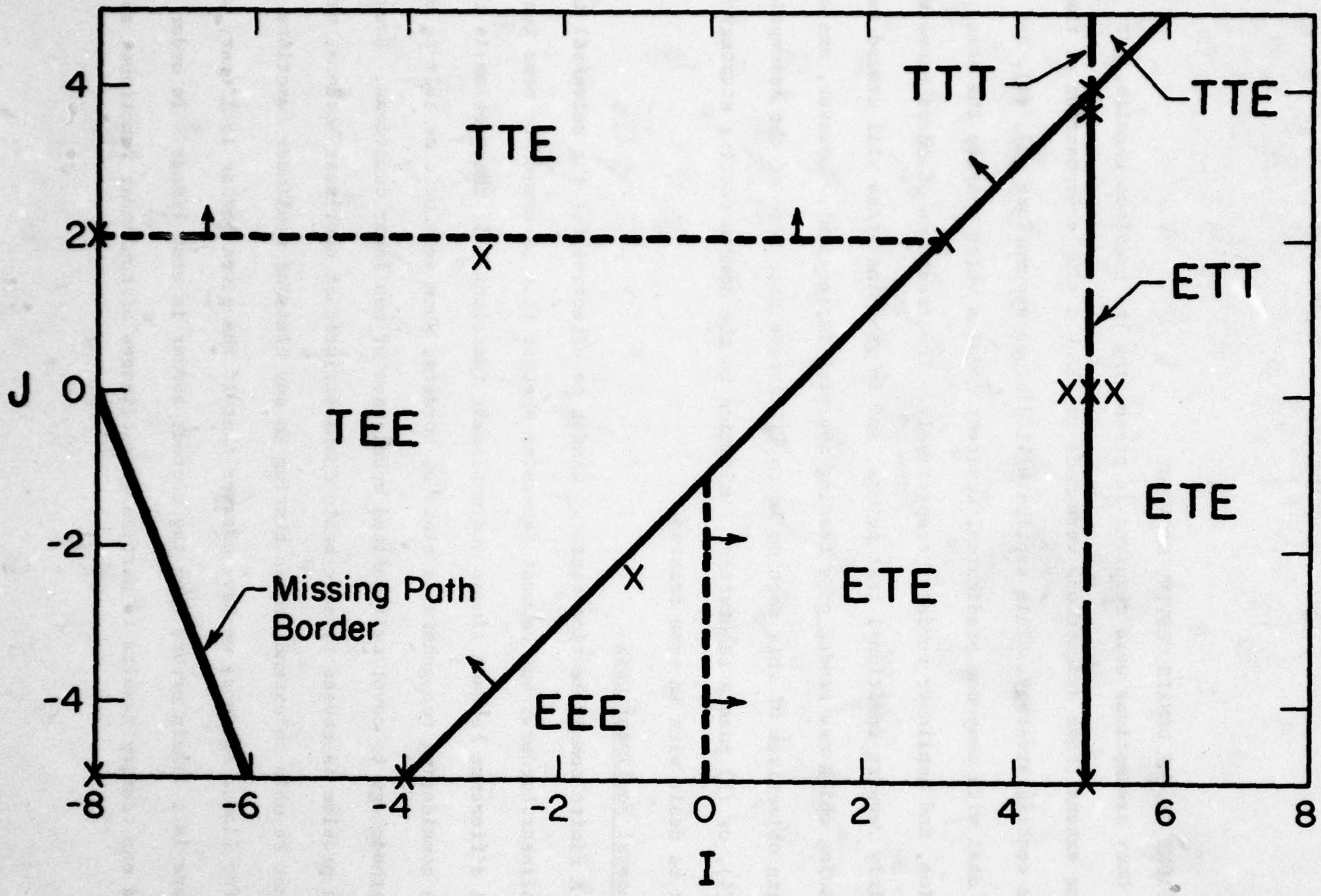


FIGURE 10 Correct Input Space for a Missing Path Error

CHAPTER 5

EXTENSIONS OF THE DOMAIN TESTING STRATEGY

Many assumptions were required in presenting the previous results, but to some extent these assumptions were made to allow a simple exposition of the domain testing strategy. This section will discuss assumptions (3), (4), and (5) which deal with compound predicates, adjacent domains which compute the same function, and nonlinear borders, respectively. The treatment of these cases will certainly require additional test points, and in some instances will demand extra processing which may render this testing approach impractical. However, one of the main objectives of this section is to illustrate that none of the assumptions (3), (4), or (5) pose a theoretical limitation to the domain testing strategy which cannot be dealt with in some fashion.

5.1 The General Nonlinear Case

A finite domain testing strategy cannot be effective for the universal class of nonlinear borders, but we must determine whether this is caused by some fundamental difference between linear and nonlinear functions. If the problem is that we are considering too general a class of borders, then we should be able to extend the methodology to cover well-defined subclasses of nonlinear functions. However, if the problem is caused by some basic characteristic of nonlinear borders, we will not be able to extend domain testing to any class of nonlinear functions.

For linear borders, we have assumed that if the given border is linear, and if there is a domain error, then the correct border is also linear. In order to extend our testing results to particular subclasses of nonlinear functions, such as quadratic or cubic polynomials, we must assume that if the given nonlinear border is in error, then the correct border is in the same nonlinear class. This nonlinear class will be specified by K parameters; for example, consider the general form of a two-dimensional quadratic in terms of variables X and Y, where A, B, C,... are coefficients, and K = 6:

$$AX^2 + BY^2 + CXY + DX + EY + F = 0.$$

Then (K-1) points can be chosen in order to solve for these K coefficients. For the example above, the five points $[X_i, Y_i]$, i = 1,...,5, should satisfy the following system of equations:

$$\begin{bmatrix} X_1^2 & Y_1^2 & X_1Y_1 & X_1 & Y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_i^2 & Y_i^2 & X_iY_i & X_i & Y_i & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_5^2 & Y_5^2 & X_5Y_5 & X_5 & Y_5 & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Define an <u>independent set</u> of (K-1) points $[X_i, Y_i]$ as a set which can be used to solve for the coefficients, and thus determine a specific member of the nonlinear class.

We can now formulate the general nonlinear domain testing strategy in terms of these observations. (K-1) ON-OFF pairs of points are chosen such that the (K-1) ON points are independent and each OFF point is chosen a distance ε from the corresponding ON point. This requires 2*(K-1) test points per nonlinear border. The (K-1) ON-OFF line segments formed by this set of pairs have been chosen so that the only correct borders which yield correct test results must intersect each of these ON-OFF line segments. For any particular correct border, there are (K-1) independent intersection points, which determines the border completely. Note that the intersection points are independent if ε is chosen sufficiently small, since the ON points are independent for the given border. A further requirement, as in the linear case, is that all OFF points satisfy all inequality borders other than the one being tested.

While a single OFF point was sufficient in the linear case, the independence criterion requires (K-1) OFF points for each nonlinear border. In the former case linearity allowed the OFF point to be shared by all the ON points, since the linear independence of the points identified as lying on the true border is guaranteed by the linear independence of the ON points themselves. If we were to test a nonlinear border with (K-1) ON points and a single OFF point, we would be able to conclude that the correct and given borders intersect at (K-1) points. However, we cannot conclude that these (K-1) points are independent. We know of no selection criterion for the ON points which would guarantee the independence of the intersection points using only one OFF point. So an effective strategy requires the full set of 2*K test points, and unfortunately K grows very rapidly as the dimensionality and degree of nonlinearity of the border increases.

A two-dimensional nonlinear border is a very special case, and even though the general strategy is effective, a slightly different testing strategy can be formulated to reduce the number of required test points. The basic difference is that the intersection between two-dimensional nonlinear borders from the same class is a finite set of points, the maximum number of which can be determined from the form of the function. For example a pair of two-dimensional quadratic curves can intersect in at most four points. This means that any set of more than four points cannot possibly lie on two distinct quadratics, and any five points uniquely determines a specific quadratic. Therefore, we do not have to worry about independence in the two-dimensional case, since any set of (K-1) distinct points will produce a system of independent linear equations. For example, any three distinct points can lie on at most one circle, since two circles cannot have more than two points in common.

We test a two-dimensional nonlinear border with K points, e.g., six for a quadratic selected in an ON-OFF-ON-OFF.... sequence along the border as diagrammed for the closed border in Figure 11. Since the correct border must pass on or above the given border at each ON point, and must pass below each OFF point, the two borders must intersect an odd number of times, let us assume once, in each ON-OFF and OFF-ON interval along the border. The K test points define (K-1) intervals on the border, each of which must contain at least one intersection point. We have shown that these (K-1) points must be independent, and since they cannot lie on two distinct borders, the given border must be correct within ε. As a technical detail, it is also possible that the correct border may be tangent to the given border at an ON point, but if this occurs, an argument involving the derivatives of the two borders at that point can be invoked to justify the choice of the test points for this two-dimensional case.

Although the domain strategy has been extended to nonlinear boundaries, points must be generated in a domain defined by nonlinear boundaries, requiring the solution of nonlinear systems of equations. Since this probably requires excessive processing for arbitrary nonlinear borders, it does not represent a very practical approach.

5.2 Adjacent Domains Which Compute the Same Function

If two adjacent domains compute the same function, any test point selected for their common border is ineffective, since the same output values are computed for the test point regardless of the domain in which it lies. We will demonstrate how domain testing can be modified to deal with this problem.

In Figure 12(a), assuming domain D_1 were being tested, we must compare the functions calculated in domains D_1 and D_2 for test point A, D_1 and D_4 for B, and D_1 and D_3 for C. One of the major problems to be solved is the identification of these adjacent domains. We assume that when testing domain

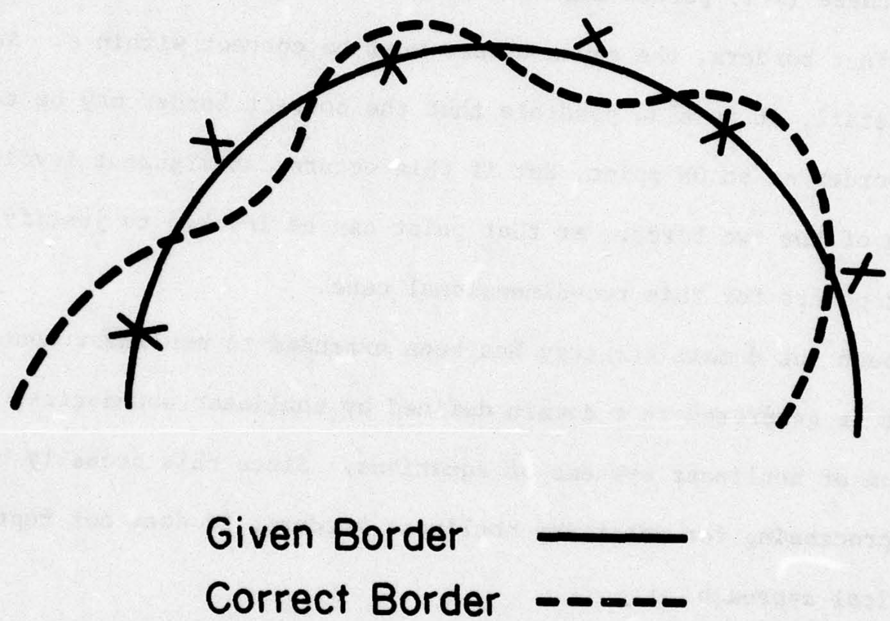


FIGURE 11 Testing a Two-Dimensional Nonlinear Border

D_1 the partitioning structure of the adjacent domains and the program paths associated with these domains is not known. It would be very complicated to have to generate the domains which are adjacent to the border being tested.

Figure 12(b) illustrates an approach to this problem. The border being tested is shifted parallel by a small distance ε, so that test points A and B now belong to adjacent domains, D_2 and D_4, respectively. The modified program is then retested using test points A and B, which will as a by-product identify the paths associated with these two adjacent domains. We can then compare the output for each test point before and after the shift. If it is different, then we can definitely conclude that the adjacent domain computes a different function, and this test point can safely be used. If the output is the same for that test point, then we can conclude that either assumption (1) or (4) is violated. However, there is no way to decide this, and the only practical approach is to use further test points. If we know that coincidental correctness cannot occur, then we could conclude on the basis of a single point that the adjacent domain computes the same function.

If two adjacent domains such as D_1 and D_2 in Figure 12(a) are found to compute the same function, then in order to carry out the domain testing strategy on the given border, new test points may have to be selected. For example, point A can no longer be used, and this requires ascertaining the border structure between D_1 and D_2. Thus a considerably amount of processing is required which is probably not practical.

In summary, a technique of testing each point twice will assure us that assumption (4) is valid, and this redundancy might be viewed as a reasonable price to pay to eliminate this restriction. However, if an instance is found where the assumption is not valid, a basic theoretical problem exists.

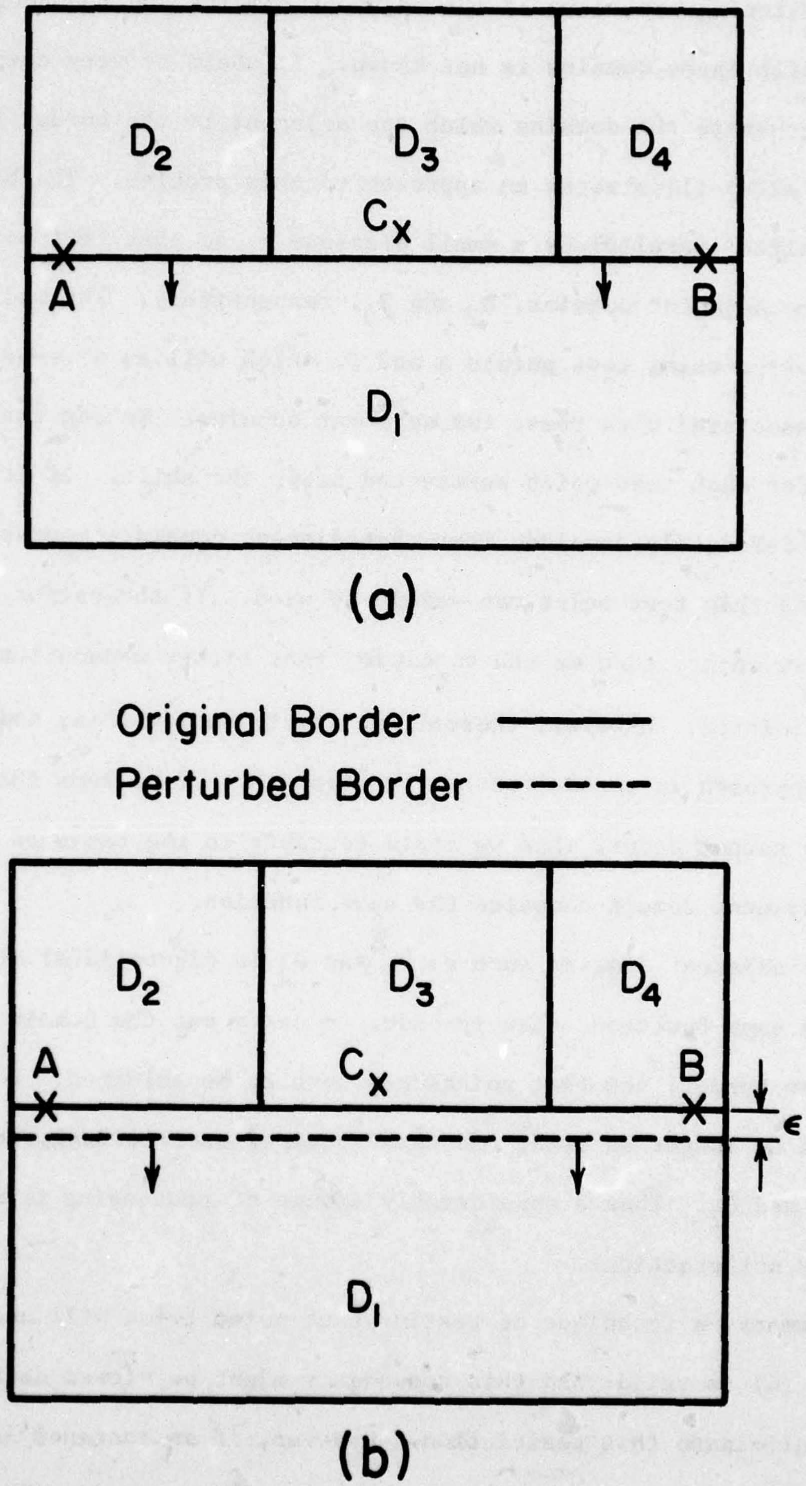


FIGURE 12 The Identification of Adjacent Domains

5.3 Domain Testing for Compound Predicates

Assumption (3) stated that a path contained only simple predicates, and this implied that the set of input points could be characterized quite simply as a single domain. We must consider what complications can occur for compound predicates, and how the domain strategy can be generalized to test paths containing these predicates.

The set of inputs corresponding to a path is defined by the path condition, consisting of the conjunction of the predicates encountered along the path. If a compound predicate of the form [C(i) AND C(i+1)] is encountered on the path, the path condition is still a single conjunction of simple predicates, and the only difference is that two of the simple predicates are produced as a single branch point on the path. No modifications of the domain testing strategy are required in this case.

However, compound predicates using the Boolean operator OR are more complicated. Consider a path containing the following predicates:

$$C_1,\ C_2,\ \ldots,\ [C_i \text{ OR } C_{i+1}],\ \ldots\ C_t$$

The path condition in this case is the conjunction of these predicates, and in standard disjunctive normal form:

$$[C_1 \text{ AND } C_2 \text{ AND } \ldots \text{ AND } C_i \text{ AND } \ldots \text{ AND } C_t]$$
$$\text{OR}\quad [C_1 \text{ AND } C_2 \text{ AND } \ldots \text{ AND } C_{i+1} \text{ AND } \ldots \text{ AND } C_t]$$

The set of input data points following this path consists of the union of two domains, each defined by the conjunction of simple predicates, and in general any number of these domains are possible.

Assuming linear predicates, each of these domains is a convex polyhedron, but the domains may overlap in arbitrary ways. The major problem caused by these compound predicates is that the domains correspond to the same path, and the assumption that adjacent domains do not compute the same function is violated. We identify three cases of importance: domains which do not overlap, domains which partially overlap, and domains which totally overlap.

The first case is indicated in Figure 13(a), where domains D_1 and D_2 are defined by the compound predicate [C_1 OR C_2], and domain D_3 corresponds to some other path. In this case our methodology can be applied to each domain separately, since the two domains for this path are not adjacent.

In Figure 13(b), the domains partially overlap, where $D_1 \cup D_2$ is the domain defined by C_1, and $D_1 \cup D_3$ is the domain defined by C_2. In the example we cannot test the domains separately, since they are adjacent and the same function is computed in each. For example, any test point for C_1, selected along that part of the border between D_1 and D_3, is ineffective since the same results are computed for it in both of these regions. So, in this case we must insure that the adjacent domain assumption is satisfied by selecting test points for C_1 and C_2 which lie in that part of the border adjacent to a domain for some other path.

In order to deal effectively with this case, some extra analysis will have to be made, first in order to identify this second case, and also to identify the actual domain, which is no longer convex. The borders of this domain are shown in bold face in Figure 13(b). This is probably no longer a practical approach, especially for higher dimensions.

The third case is shown in Figure 13(c), where the domain D_1 for predicate C_1 is a subset of the other domain, $D_1 \cup D_2$, which is obtained for predicate C_2. This presents a serious problem since there are no test points for border B of domain D_1 which can satisfy the adjacent domain assumption, and therefore B cannot be tested effectively. The technique developed in the previous section should help to identify this case. However, even if this case could be identified, testing for border B is no longer a practical procedure.

So, in summary, a compound predicate of the form [C1 AND C2] is the same as two simple predicates, and domain testing can be applied to a domain defined with this type of compound predicate. In addition, if the compound predicate

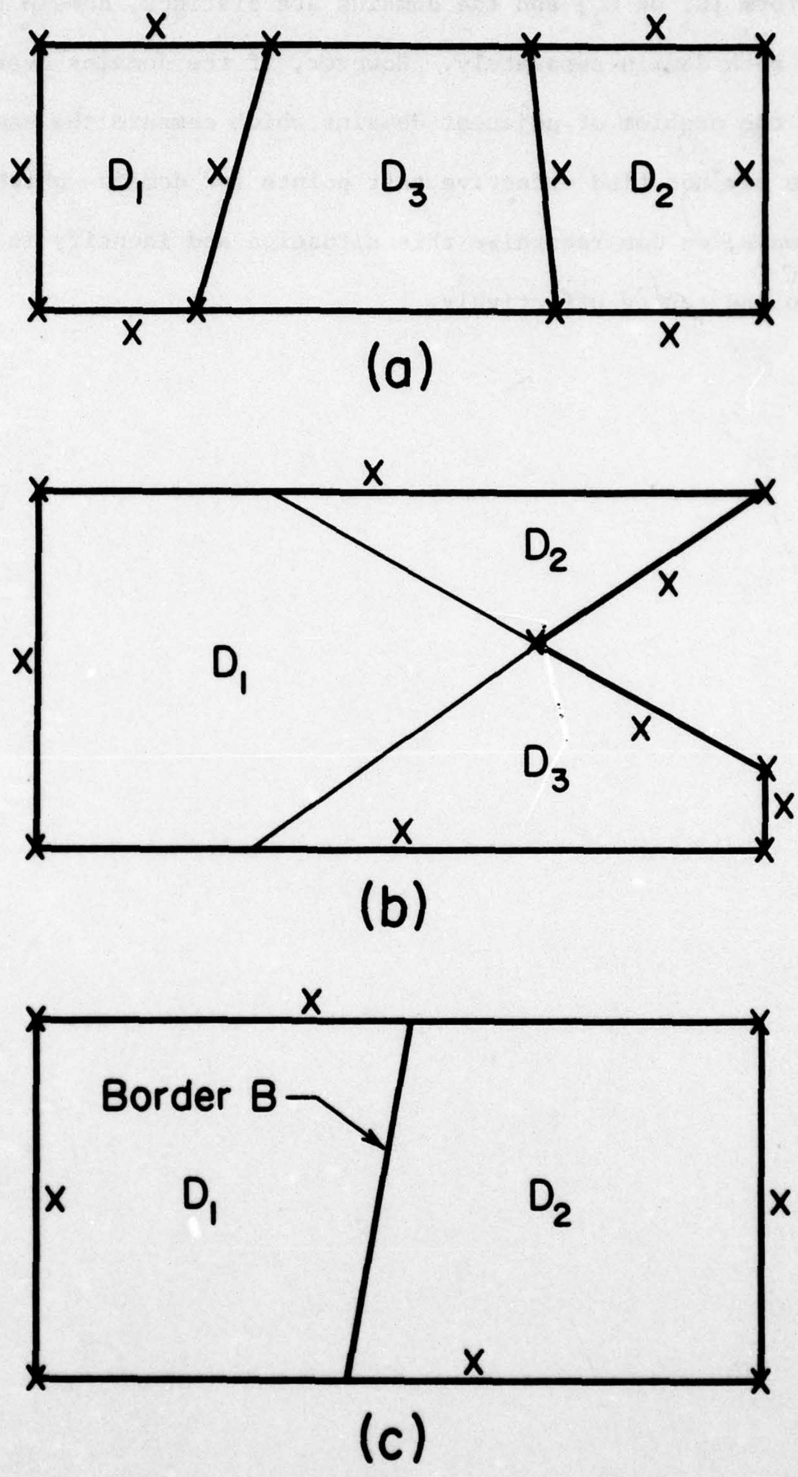


FIGURE 13 Domains Defined With Compound OR Predicates

is of the form [C_1 OR C_2] and the domains are distinct, domain testing can be applied to each domain separately. However, if the domains overlap, this introduces the problem of adjacent domains which compute the same function. Although we may not find effective test points for domains which overlap in arbitrary ways, we can recognize this situation and identify it as a border which cannot be tested effectively.

CHAPTER 6

ERROR ANALYSIS OF DOMAIN BORDERS AND DISCRETE SPACES

An error analysis of domain borders is needed to resolve the following questions:

i) How small should ε be chosen in selecting an OFF test point for linear borders, and where are optimal locations for the test points?

ii) We required the OFF test point for a given border to satisfy all inequality borders except that being tested; how do potential errors in other borders of the domain affect this requirement?

iii) What are the difficulties in applying domain testing in a discrete space or in a space in which numerical values can only be represented with finite resolution, and can these difficulties be circumvented by taking reasonable precautions with the method?

These and other error analysis problems are dealt with in detail in reference [12]. It is interesting to observe that the answers to questions i), ii), and iii) all involve the same worst-case situation: when two adjacent linear borders of the same domain are nearly parallel. Figure 14 indicates the two cases which can arise from adjacent linear borders which are nearly parallel. Figure 14(a) shows a given border segment EF in which the two adjacent border segments EP and FQ both make large <u>external angles</u> θ_1 and θ_2, near 180°, with the given border EF. This leads to very small supplementary <u>internal angles</u> ϕ_1 and ϕ_2, and especially for ϕ_2, this results in a very sharp "corner" of the domain. In Figure 14(b), the adjacent borders PE and FQ are again nearly parallel to the given border EF, but a different case is created. In this case, external angles θ_1 and θ_2 are very small, and the internal angles ϕ_1 and ϕ_2 are both near 180°.

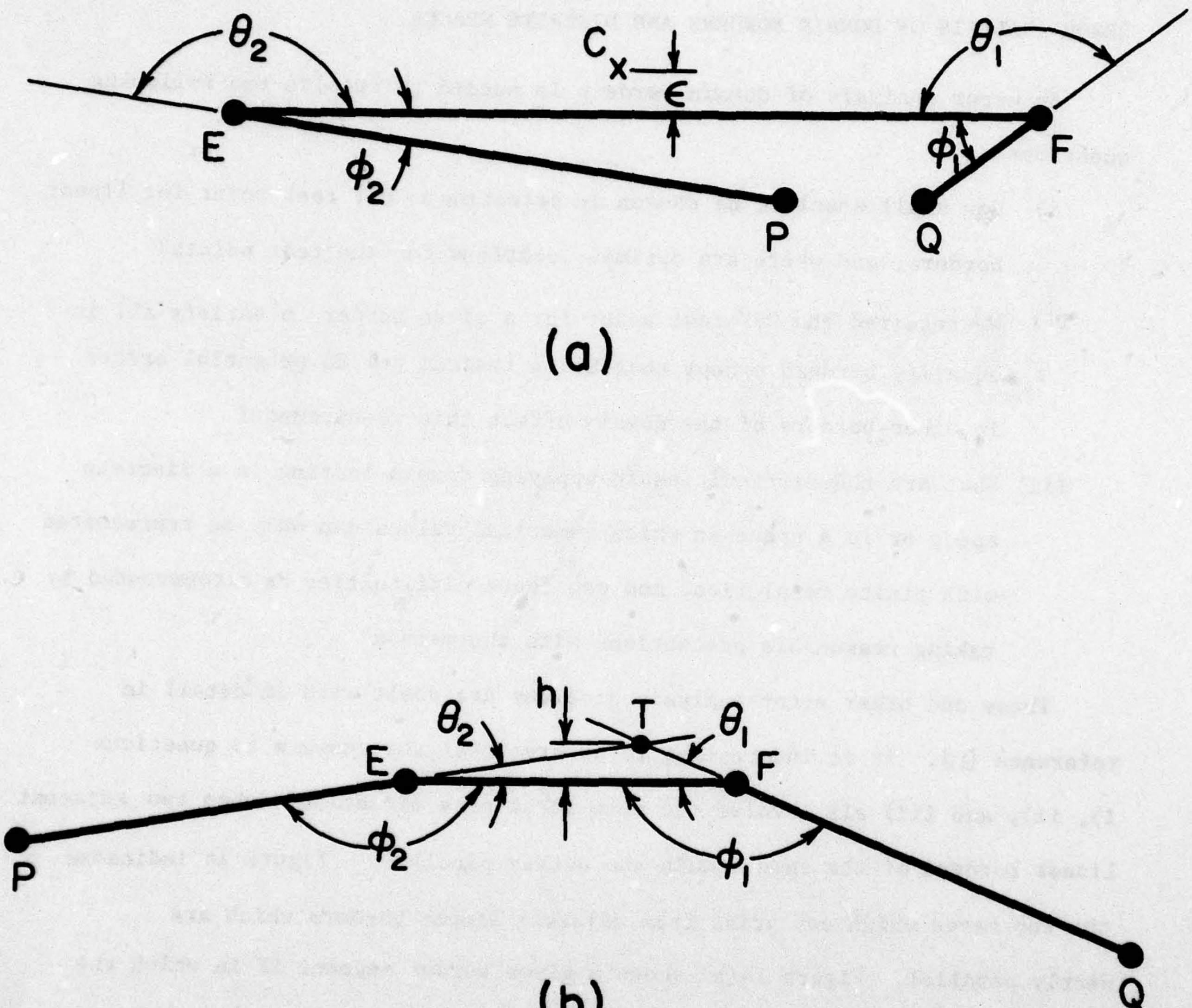


FIGURE 14 Adjacent Border Segments Which are Nearly Parallel

We will briefly argue in this report that one of these two situations is the key to the analysis of questions i), ii), and iii), and we refer the reader to reference [12] for further details and proofs. Section 6.1 introduces an error measure which will indicate the best location for each of the three test points. Section 6.2 will deal with the problem of how interacting border changes may affect the location of the test points. Section 6.3 briefly introduces the problem of domain testing in discrete spaces, and gives a sufficient condition to guarantee effective test points can always be chosen. Since all the above arguments are given only for two dimensions, Section 6.4 will show that the same basic approach is effective for higher dimensions.

6.1 An Error Measure for Test Point Selection

In Figure 14(a), consider the selection of three test points A, B, and C for testing border segment EF. It is shown in reference [12] that the best positions for two of them, say A and B, are points E and F, so the remaining problem is the location of test point C. We have observed that if the given border EF is in error, then test points A, B and C will fail to detect errors if the correct border is one which intersects line segments AC and BC. Thus given C which is at a distance ε from the given border and halfway between A and B, an appropriate error criterion could be the "number" of erroneous points which would be undetected, i.e., the <u>area between the two borders</u>, possibly limited by either or both of the extensions of the adjacent borders EP and FQ. It can be shown that this area measure can be bounded by the expression

$$\frac{\varepsilon[\overline{EF}]^2}{\overline{EF} + 2\varepsilon \cot \theta},$$

where θ is the larger of θ_1 and θ_2.

In order for this error measure to be finite, it is necessary that both θ_1 and θ_2 are not too close to 180° for given ε. If $|\cot\theta| \ll \frac{\overline{EF}}{\varepsilon}$, then the error measure is on the order of $\varepsilon \cdot \overline{EF}$. This gives some guidance as to the choice of ε for point C.

6.2 Interacting Border Segments

In presenting the domain strategy, we required the OFF test point to satisfy all inequality borders except the border being tested. Usually this does not impose much of a constraint on the choice of the OFF point, but Figure 14(b) illustrates a situation in which a severe constraint exists. We can show that

$$h = \frac{\overline{EF}}{(\cot\theta_1 + \cot\theta_2)},$$

and since $\varepsilon < h$ for choosing the OFF test point, this again shows the effect if either θ_1 or θ_2 or both are very small.

The same situation applies for interacting adjacent borders, and is illustrated in Figure 15. As long as the OFF points C_1 and C_2 for each of the adjacent borders are chosen sufficiently close to those borders, and the external angles θ_1 and θ_2 are not too small, then the adjacent borders have a minimal influence on the selection of the OFF point C for border EF. For example, point C must lie inside triangle EFU determined by given borders EP and FQ. The correct borders which pose the worst case in limiting the selection of point C are shown as dashed lines in Figure 15; these limiting correct borders are determined by how close C_1 and C_2 have been chosen to their respective test borders. As a result of these conditions, point C is constrained to lie within triangle EFV, a more restrictive condition than presented by triangle EFU. It should be clear that if either θ_1 or θ_2 is too small, or either C_1 or C_2 is chosen too far from its respective test border, the region from which C could be chosen would become restrictively small.

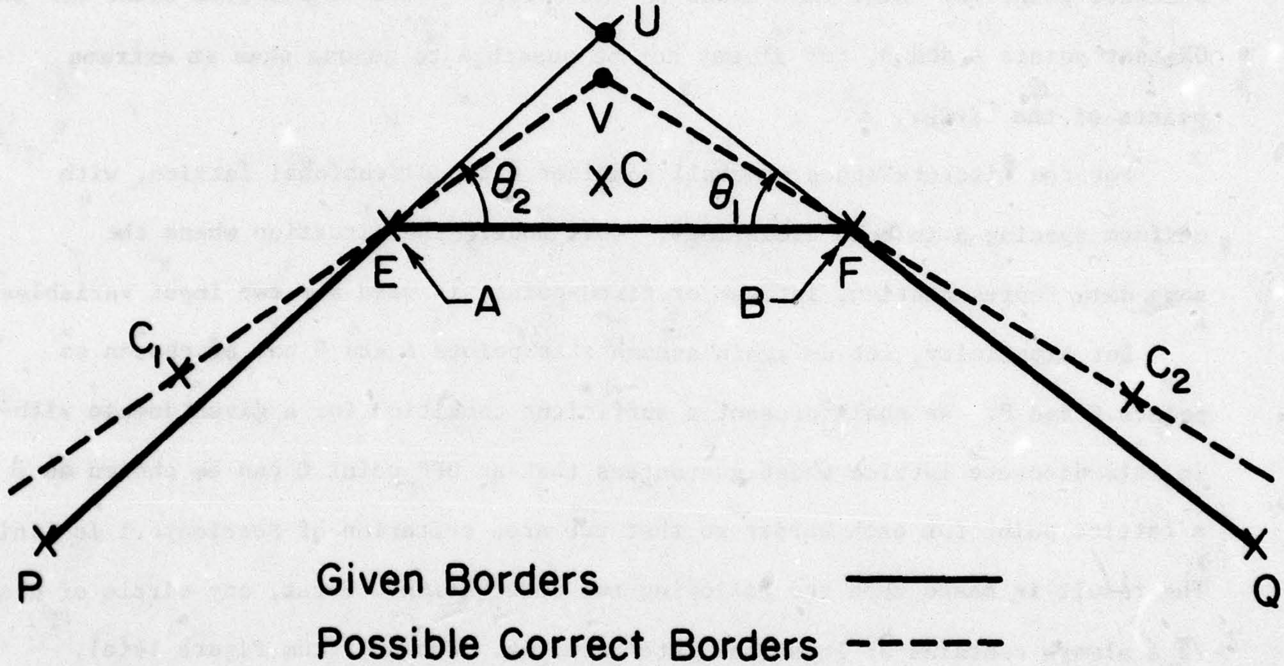


FIGURE 15 Effect of Interacting Adjacent Borders on Test Point C

6.3 Discrete Space Analysis

The previous several sections have indicated that if adjacent borders are nearly parallel, then test point C is required to lie very close to the border being tested. But in a discrete space this could cause a severe problem, for no discrete point may exist that close to the border. Similar problems exist for the ON test points A and B, for it may not be possible to choose them at extreme points of the border.

For the discrete space we shall consider a two dimensional lattice, with uniform spacing Δ in both dimensions. This models the situation where the same data representation, integer or fixed-point, is used for two input variables.

For simplicity, let us again assume that points A and B can be chosen as points E and F. We shall present a sufficient condition for a given domain within this discrete lattice which guarantees that an OFF point C can be chosen as a lattice point for each border so that the area criterion of Section 6.1 is finite. The result is based upon the following two observations. First, any circle of diameter $\sqrt{2}\,\Delta$ always contains at least one lattice point. Second, from Figure 14(a), note that if either external angles θ_1 or θ_2 are too near 180°, then the "width" of the domain will tend to be very small in terms of the lattice resolution Δ.

More formally, define the *diameter* d of a convex polygonal domain to be the shortest distance from any extreme point to any domain edge not adjacent to that extreme point; this corresponds to our informal argument about domain "width". The sufficient condition can then be stated as:

Proposition 5

Given a domain with diameter d in a lattice with resolution Δ, if

$$d > \left(\frac{3}{\sqrt{2}}\right) \Delta = (2.12)\ \Delta,$$

then a lattice OFF point can be chosen for every border, and moreover all external angles θ_1 and θ_2 are constrained by

$$\left|\cot \theta_1 + \cot \theta_2\right| < \frac{\overline{EF}}{\left(\frac{3}{\sqrt{2}}\right)\Delta} .$$

It is clear that there are some domains in a discrete space which cannot be tested, but these are pathological cases where one of the domain dimensions is on the order of the lattice resolution. Moreover, the result indicates a simple computation in terms of the domain diameter to determine when such domains are presented for testing. For domains which can be tested in a discrete space, the important result from Proposition 5 is that a restriction has been obtained on angles θ_1 and θ_2 which precludes both angles which are close to 180° and angles which are too small.

6.4 Extensions of Error Analysis to Higher Dimensions

The previous arguments have all been made for two dimensions, so it is important that the essential ideas can be generalized to higher dimensions. We can observe that if two border segments are adjacent, they are intersecting hyperplanes. Again, problems may arise if these two hyperplanes H_1 and H_2 are nearly parallel, and this can be measured by taking the inner product of their unit normal vectors $\hat{n}_1$ and $\hat{n}_2$, yielding the cosine of the angle α between them:

$$\cos \alpha = \hat{n}_1 \cdot \hat{n}_2$$

Consider Figure 16 which indicates the testing strategy for three dimensions. H_1 is assumed to be the border to be tested by ON points A_1, A_2, A_3 and C is the OFF point. H_2 is an adjacent border nearly parallel to H_1, and H_1 intersects H_2 at line L. If it is suspected that C may not be chosen close enough to H_1, only those borders which make an angle α of 10° or less with H_1 need to be investigated further.

To determine a test point C, we need to select that correct border hyperplane which is the worst case relative to border H_2, and then determine whether or not these two hyperplanes intersect. This computation is quite straightforward, and the following algorithm together with Figure 16 should indicate how it can be accomplished:

(a) select the ON point A_i furthest from line L (this is A_3 in Figure 16); the worst case correct border hyperplane H_3 is then determined by line L and line segment A_iC;

(b) drop a perpendicular line segment from A_i to line L; this makes an angle β with line segment A_iC', where C' is the projection of point C down on the hyperplane H_1 being tested; recall that C' is known, for point C is obtained by first finding C';

(c) the angle ϕ between H_1 and H_3 can be found by

$$\tan \phi = \frac{\varepsilon}{\overline{A_iC'} \cos \beta} ;$$

(d) if $\phi < \alpha$, then hyperplanes H_2 and H_3 do not intersect; otherwise, ε should be chosen smaller so that this condition is satisfied.

Again, in this analysis, the fact that adjacent borders H_1 and H_2 are nearly parallel proves to be the key point in selecting test point C. Yet, the above algorithm can be used to choose C so as to compensate for this condition.

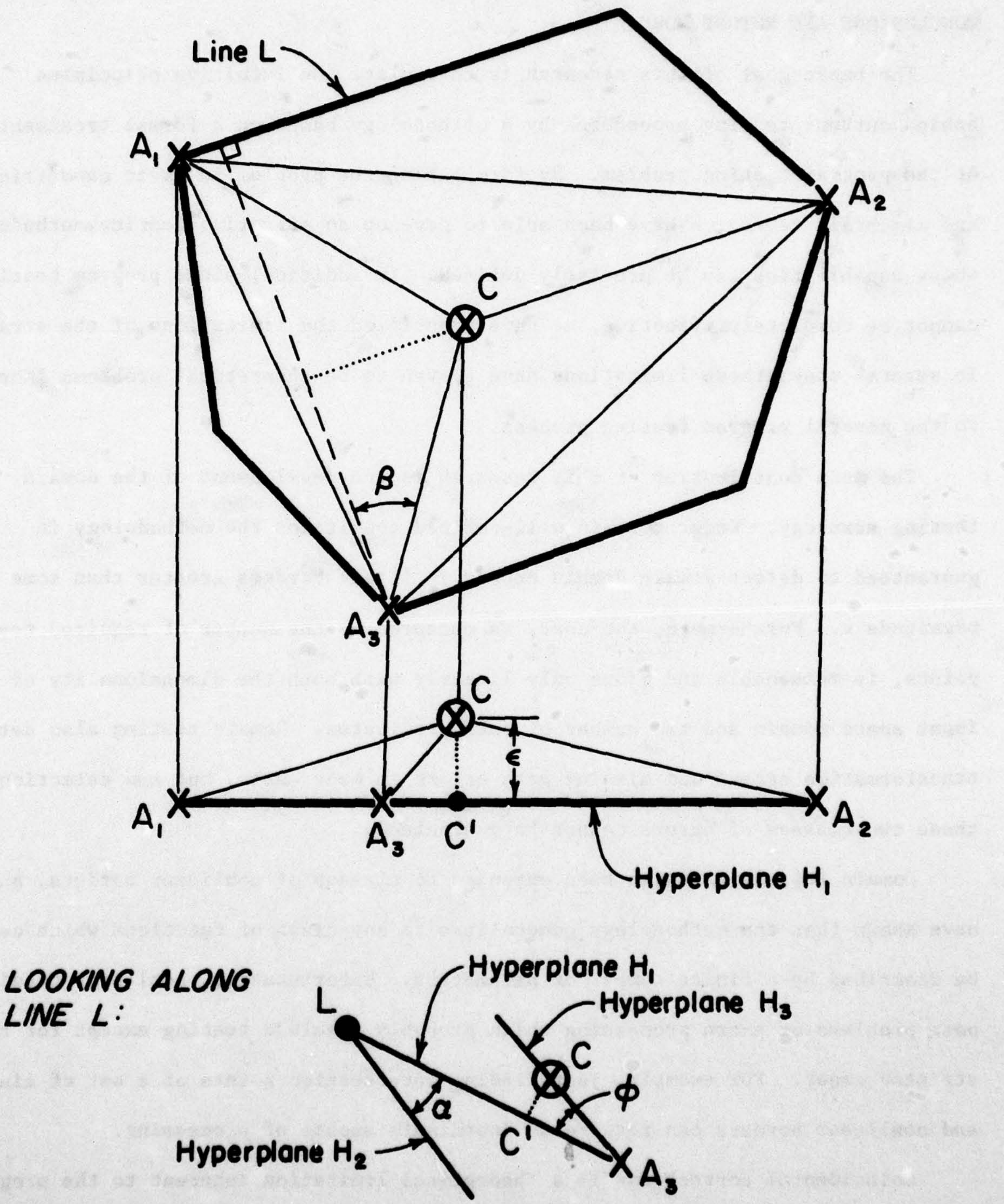


FIGURE 16 Error Analysis in Three Dimensions

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The basic goal of this research is to replace the intuitive principles behind current testing procedures by a methodology based on a formal treatment of the program testing problem. By formulating the problem in basic geometric and algebraic terms, we have been able to develop an effective testing methodology whose capabilities can be precisely defined. In addition, since program testing cannot be completely effective, we have identified the limitations of the strategy. In several cases these limitations have proven to be theoretical problems inherent to the general program testing process.

The main contribution of this research is the development of the domain testing strategy. Under certain well-defined conditions the methodology is guaranteed to detect domain domain errors in linear borders greater than some small magnitude ε. Furthermore, the cost, as measured by the number of required test points, is reasonable and grows only linearly with both the dimensionality of the input space domain and the number of path predicates. Domain testing also detects transformation errors and missing path errors in many cases, but the detection of these two classes of errors cannot be guaranteed.

Domain testing has also been extended to classes of nonlinear borders, and we have shown that the methodology generalizes to any class of functions which can be described by a finite number of parameters. Unfortunately, nonlinear predicates pose problems of extra processing which probably preclude testing except for restricted cases. For example, just finding intersection points of a set of linear and nonlinear borders can require an inordinate amount of processing.

Coincidental correctness is a theoretical limitation inherent to the program testing process, and we have argued that it prevents any reasonable finite testing procedure from being completely reliable. In particular, the possibility of coincidental correctness means that an exhaustive test of all points in an input domain is theoretically required to preclude the existence of computation errors on a path. Within the class of all computable functions there exist functions which coincide at an arbitrarily large number of points, but if there is sufficient resolution in the output space, coincidental correctness should be a rare occurrence for functions commonly encountered in data processing problems.

The class of missing path errors, particularly those of reduced dimensionality, has proven to be another theoretical limitation to the reliability of any finite testing strategy. Although our methodology cannot be guaranteed to detect all instances of this type or error, it can be extended to detect some well-defined subclasses of missing path errors. Unfortunately, the extra cost of this modification may be unacceptably high. Our analysis of missing path errors has shown that the cause of the difficulty is that the program does not contain any indication of the possible existence of a missing path error. Therefore, without additional information, a reasonable testing strategy for this class of errors cannot be formulated.

The domain testing strategy requires a reasonable number of test points for a single path, but the total cost may be unacceptable for a large program containing an excessive number of paths. In particular, this may occur for large programs with complicated control structures containing many iteration loops. Additional research is needed to substantially reduce the number of potential paths. One area being investigated takes advantage of the fact that program modules are often independent in that the control flow of one does not depend upon variables defined in the other. In this way the combinatorial growth of the number of domains to be tested can be controlled, and the domain strategy can be made more practical. It remains to be shown to what extent this independence property can be applied, and experimental evidence is needed of how frequently independent modules occur in widely available programs.

We have assumed that an "oracle" exists which can always determine whether a specific test case has been computed correctly or not. In reality, the programmer himself must make this determination, and the time spent examining and analyzing these test cases is a major factor in the high cost of software development. One possible avenue for future research would be to automate this process by using some form of input-output specification. If the user provides a formal description of the expected results, the correctness of each test case can be decided automatically by determining whether the output specification is satisfied. This would reduce the cost of testing tremendously, and these new testing techniques would gain acceptance more quickly since the tedious task of verifying test data would be eliminated. In addition, any extra information supplied by the user might be useful in specifying special processing requirements which would indicate the existence of a possible missing path error.

The domain test strategy is currently being implemented, and will be utilized as an experimental facility for subsequent research. Experiments should indicate what sort of programming errors are most difficult to detect, and should yield extensive dynamic testing data. A most important contribution would be to indicate both programming language constructs and programming techniques which are easier to test, and thus would produce more reliable software.

1. Boyer, R.S., Elspas, B., and Levitt, K.N., "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution", PROCEEDINGS-1975 International Conference on Reliable Software, Los Angeles, Ca., April 1975, 234-245.

2. Clarke, L.A., "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, Sept. 1976, 215-222.

3. Cohen, E.I. and White, L.J., "A Finite Domain-Testing Strategy for Computer Program Testing", Technical Report 77-13, Computer and Information Science Research Center, The Ohio State University, August, 1977.

4. Cohen, E.I., "A Finite Domain-Testing Strategy for Computer Program Testing", Ph.D. Dissertation, The Ohio State University, June, 1978.

5. Elshoff, J.L., "A Numerical Profile of Commercial PL/I Programs", Report No. GMR-1927, Computer Science Department, General Motors Research Laboratories, Warren, Mich., Sept. 1975.

6. Elshoff, J.L., "An Analysis of Some Commercial PL/I Programs", IEEE Transactions on Software Engineering, Vol. SE-2, No. 2, June 1976, 113-120.

7. Goodenough, J.B. and Gerhart, S.L., "Toward A Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, 156-173.

8. Howden, W.E., "Methodology for the Generation of Program Test Data", IEEE Transactions on Computers, Vol. C-24, No. 5, May 1975, 554-560.

9. Howden, W.E., "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, Sept. 1976, 208-215.

10. Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software - Practice and Experience, Vol. 1, No. 2, April-June, 1971, 105-133.

11. Ramamoorthy, C.V., Ho, S.F., and Chen, W.T., "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, 293-300.

12. White, L.J.,Teng F.C., Kuo, H.C., and Coleman, D.W., "An Error Analysis of the Domain Testing Strategy", Technical Report 78-2, Computer and Information Science Research Center, The Ohio State University, August, 1978.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 18 AFOSR-TR-79-0444
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): 6 A DOMAIN STRATEGY FOR COMPUTER PROGRAM TESTING
5. TYPE OF REPORT & PERIOD COVERED: 9 Interim rept.
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): 10 Lee J. White, Edward I. Cohen and B. Chandrasekaran
8. CONTRACT OR GRANT NUMBER(s): 15 AFOSR-77-3416
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The Ohio State University, Computer and Information Science Research Center, Columbus, Ohio 43210
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 16 2304 17 A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332
12. REPORT DATE: 11 Aug[redacted]78
13. NUMBER OF PAGES: 74
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 12 76p.
15. SECURITY CLASS. (of this report): UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

14 OSU-CISRC-TR-78-4

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Computer programs contain two types of errors which have been identified as computations errors and domain errors. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program. A path contains a computation error when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables. A testing strategy has been designed to detect domain errors, and the conditions under which this strategy is reliable are given and characterized. A by-product of this domain strategy is a partial ability

DD FORM 1 JAN 73 1473 407586 UNCLASSIFIED

next page

20. Abstract continued.

cont. to detect computation errors.